FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Aplikace pro chytrý monitoring Docker kontejnerů
Student:	Bc. Matyáš Gallas
Vedoucí:	Ing. Marek Suchánek
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Docker je velmi oblíbený pro usnadnění nasazení aplikacích různých druhů. Ačkoliv existuje řada podpůrných nástrojů, které pomáhají monitorovat aplikace běžící v Docker kontejnerech, bývají složité na nasazení, konfiguraci a nasazení, či jsou neintuitivní z důvodu mnoha rozličných funkcí. Cílem této práce je navrhnout a implementovat nástroj, který bude zaměřen na snadnost nasazení i použití a který bude umožňovat sledovat základní metriky aplikací či sad aplikací běžících v Docker.

- Analyzujte fungování Docker a možnosti sledování základních metrik. Popište také možnosti a rozdíly při použití Docker Swarm.
- Proveďte stručnou rešerši existujících řešení sledování metrik v Dockeru.
- Navrhněte vlastní aplikaci složenou s dílčích komponent pro sledování metrik, jejich ukládání, konfiguraci a zasílání upozornění.
- Dle návrhu aplikaci implementujte, otestujte a zdokumentujte.
- Zhodnoťte přínosy aplikace z pohledu softwarového inženýrství a stručně porovnejte s existujícími řešeními.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 7. října 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Aplikace pro chytrý monitoring Docker kontejnerů

Bc. Matyáš Gallas

Katedra softwarového inženýrství
Vedoucí práce: Ing. Marek Suchánek

4. května 2022

Poděkování

Nejprve bych rád poděkoval svému vedoucímu Ing. Marku Suchánkovi za vřelou pomoc a zkušenosti i v případě diplomové práce, které mi byly velkým přínosem. Mé poděkování současně patří mé blízké rodině, která za mnou neustále stála a byla mi podporou, a také mým přátelům za cenné rady.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 4. května 2022 . . . . . .

Odkaz na tuto práci

Gallas, Matyáš. *Aplikace pro chytrý monitoring Docker kontejnerů*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Tato diplomová práce si klade za cíl navrhnout snadno nasaditelné a konfigurovatelné řešení umožňující monitorování metrik Docker kontejnerů a vše co s monitorováním souvisí. Bude se postupovat dle tradičních metod vývoje softwaru. Nejprve tato práce seznámí čtenáře s problematikou sledování kontejnerů, nastíní jejich fungování a činnosti platformy Docker. Následně se v analýze zaměří na to jakým způsobem lze nejlépe kontejnery sledovat. Dále v návrhu vyřeší výběr vhodných technologií pro zajištění výkonosti v kontrastu s nízkými nároky na zdroje. Na analýzu a návrh navazuje praktická část, která se zabývá vývojem samotného systému, který umožní monitorování. Výsledkem je funkční systém, který lze jednoduše nasadit a dále rozšiřovat dle potřeb.

Klíčová slova informační systém, webová aplikace, monitorování, Docker, analýza, sledování prostředků, kontejner, sledování metrik, Spring framework, .Net, Blazor, C++, Java, QuestDB, docker-compose, Cgroups

Abstract

This diploma thesis aims to design a configurable, easy-to-deploy system that allows users to monitor Docker container metrics. In the first part of the thesis, readers are introduced to the challenges of monitoring, the concept of containers, and the Docker platform. The subsequent chapters describe the development process, which follows traditional software development methods. The analysis chapter mainly focuses on finding the best solution for collecting the metrics from Docker. The design chapter addresses the selection of appropriate technologies to ensure performance whilst maintaining low resource requirements. Lastly, the technical implementation of the system is summarized. The result is a functional, flexible system that can be easily deployed and expanded as needed.

Keywords information system, web application, monitoring, Docker, analysis, metrics monitoring, container, Spring framework, .Net, Blazor, C++, Java, QuestDB, docker-compose, Cgroups

Obsah

Seznam obrázků

Úvod

Bez počítačových aplikací se dnes neobjede téměř žádná společnost či instituce. S rostoucí komplexitou procesů a nárůstem počtu společností, poptávka po aplikacích stále roste. Není tak divu, že aplikací je na světě čím dál víc a stále častěji je kladen důraz na jejich optimalizaci, škálovatelnost a správné využití přidělených zdrojů.

Jedním z trendů dnešní doby v oblasti aplikací je bezpochyby technologie kontejnerů, která přináší řadu výhod mimo jiné izolaci aplikace se všemi jejími knihovnami, usnadnění portability nebo ulehčení škálovatelnosti. Nejvíce výhod poskytuje použití této technologie vývojářům mikroservise, ale také ulehčí práci všem, co pracují se softwarem. V současnosti tuto technologii využívají převážně IT společnosti, ale každým rokem přitahuje tato technologie kontejnerů více a více příznivců. Jednou z bezpochyby významných a oblíbených platforem pro kontejnerizaci je řešení Docker.

Při vývoji aplikací nás čeká v určité fázi ladění, potencionální hledání závad či optimalizace. V těchto situacích nám může velice pomoci sledování různých metrik v čase, jako vytíženosti procesoru, komunikace na síti nebo využití paměťových prostředků. Monitorování tak potencionálně zrychlí odhalování závad, které by potencionálně mohly zůstav v aplikaci a přispívat tak k vyšší náročnosti provozu či pádu aplikace jako takové.

Monitorování je ale využitelné nejen ve fázi vývoje, ale rovněž i za plného provozu aplikace v reálných podmínkách. Sledování metrik může pomoci při rozhodování přidělování prostředků nebo odhalování slabých a vytížených míst. Monitorovací software lze využít pro upozornění na neobvyklé chování aplikace pomocí notifikace a zabránit tak potencionální hrozbě vyčerpání prostředku nebo dokonce celkovému pádu aplikace.

Pro monitorování kontejnerů existuje celá řada open-source i komerčních řešení. Jejich hlavní nevýhodou je složitá konfigurace a nasazení pro jejich správnou funkci. Často obsahují rozličné množství funkcí pro uspokojení širokého spektra uživatelů a stávají se tak komplikované na používání. S vysokou

komplexností těchto řešení přichází také problém s jejich dalším rozšířením. Spousta uživatelů tak ani neuvažuje o využití výhod monitorování kontejnerů.

Nabízí se tedy možnost vytvoření nástroje, který by byl jednak jednoduchý na nasazení a zároveň snadno použitelný pro monitorování Docker kontejnerů.

Kapitola 1

Cíl práce

Hlavním cílem této práce je vytvořit nástroj pro monitorování kontejnerů, na základě kterého půjde snadno monitorovat aplikace běžící na platformě Docker s důrazem na snadnost nasazení a konfiguraci. Zpracování bude dle tradičních metod vývoje softwaru s důrazem na možnost pozdějšího rozšíření. Samotný nástroj bude možné provozovat jak na počítači, tak na serveru s předinstalovanou platformou Docker. Podmínkou pro provoz monitorování je operační systém Linux. Přístupovým bodem pro sledování bude webový prohlížeč. Implementovaný nástroj musí splňovat snadnou ovladatelnost a zabezpečení před přístupem neoprávněných osob.

Práce je zpracována dle tradičních metod softwarového inženýrství. V první části se zabývá uvedením do světa kontejnerů, analýzou způsobů jejich monitoringu a výběrem toho nejvhodnějšího. Zároveň nebude opomenut ani průzkum již existujících řešení z oblasti monitorování metrik za účelem lepšího seznámení s problematikou. Další část navazuje na zjištěné poznatky a pojednává o výběru správné technologie pro implementaci. Následuje popis použité architektury a implementace zakončená testováním. Poslední kapitola je věnovaná možnostem budoucího rozšíření a přínosem této práce.

Kapitola 2

Problematika monitorování kontejnerů

Tato kapitola nejprve seznamuje s pojmem virtualizace, a jak je s ní spojena kontejnerizace a platformy, které jí umožňují. Dále upozorňuje na důležitost shromažďování a analýzu základních metrik. V neposlední řadě uvádí jednu ze známých kontejnerizačních platforem zvanou Docker, kde se zabývá základním pohledem na architekturu a jak se dají kontejnery sledovat. Na závěr je uveden do kontextu orchestrační nástroj Docker Swarm a způsob, jak ho lze monitorovat. Ze zjištěných informací v této kapitole o monitorování se dále odvíjí vlastní implementace.

2.1 Virtualizace vs. Kontejnerizace

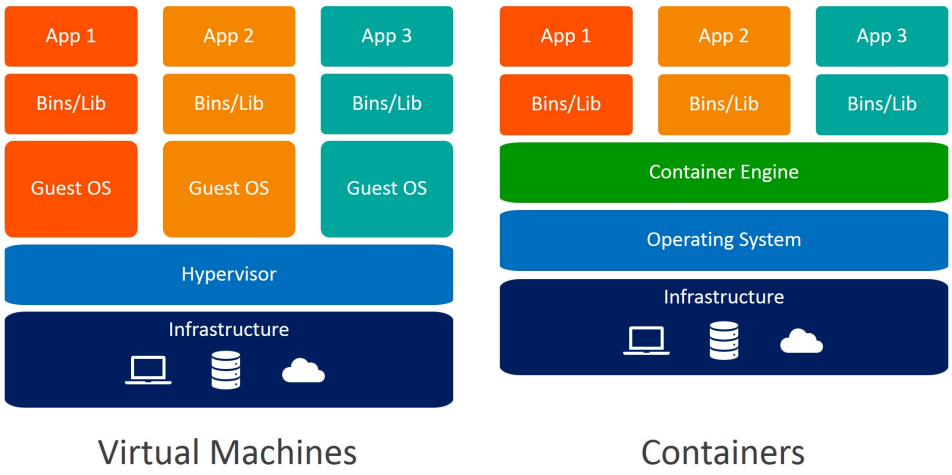
Virtualizace je přistup, kdy je aplikace odstíněna od fyzického zařízení, na kterém zrovna běží. Je vytvořená tzv. abstraktní vrstva, která odděluje tuto aplikaci od hardwaru. Tato vrstva dokáže hardwarové prostředky jako procesorový čas, paměť nebo úložiště rozdělit mezi virtualizované aplikace. Virtualizaci lze rozdělit na několik způsobů lišících se od sebe výkonem, složitostí nebo požadavky na hardware. [2]

Plná virtualizace, označovaná také jako top-down přístup, je jedním ze způsobů, při kterém se na fyzické zařízení nainstaluje komponenta zvaná hypervisor umožňující vytvářet virtuální stroje (VM). Každý VM se pak chová jako samostatné zařízení s vlastním operačním systémem nezávislým na hostitelském zařízení. Virtuální stroj se pak chová jako skutečný počítačový systém. Tento způsob je ale velice nákladný a spotřebuje velké množství výkonu v řádu desítek procent. [2]

Dalším způsobem je virtualizace jádra operačního systému zvaná kontejnerizace, která je označená jako bottom-up přístup. V tomto případě mají všechny aplikace přístup pouze k jedné instanci operačního systému, který

si sdílejí. Jsou však stále od sebe izolovány s omezeným přístupem ke zdrojům. Kontejnerizace sdružuje kód aplikace spolu se souvisejícími konfiguračními soubory, knihovnami a závislostmi, které jsou nutné pro její spuštění a abstrahuje tak kontejner od hostitelského systému. Kontejner se pak díky tomu stává přenositelný bez problému na jakoukoliv platformu. Tento způsob je méně náročný na zdroje a má lepší spolehlivost oproti úplné virtualizaci. Tato virtualizace se navíc dá jednoduše nasadit na cloud. Porovnání obou přístupů můžeme vidět na obrázku 2.1. [1, 2]

Nedá se však jednoznačně označit, který způsob je lepší. Může se zdát, že kontejnerizace je ultimátní nástroj, ale vždy záleží na případu užití. Kontejnerizace je zejména vhodná pro mikroservisy (malé aplikace), pro které je neefektivní vytvářet vlastní server. Pokud se nejedná o Software-as-a-Service, je lepší se kontejnerům vyhnout a zvážit využití VM. V této práci se ale dále zabýváme jen kontejnerizací. [4]



Obrázek 2.1: Porovnání virtuálního stroje a kontejneru převzato z [3]

2.1.1 Kontejner

Kontejnery jsou izolované prostředí (jednotka), ve kterých běží kód aplikace společně s knihovnami a závislostmi. Mají tak vše potřebné pro jejich běh již v sobě. Izolace kontejneru je docílena pomocí namespaces, které poskytují vlastní pohled na operační systém. Pro nastavení omezení prostředků (CPU, Paměť, IO) je použita funkce linuxového jádra control group (cgroup). Poskytuje dále prioritizaci nebo změnu statusu(frozen, stopped). [10] Jedná se o virtualizaci jádra operačního systému. Kontejnery jsou tak malé, rychlé a přemostitelné, jelikož nezahrnují operační systém.Využívají totiž funkce a prostředky hostitelského OS. Kontejner nemůže existovat bez image, je na něm závislý a používá ho pro vytvoření běhového prostředí a spuštění aplikace. Tento

image je možné spustit na hostovi lokálně či vzdáleně na cloudu a zformovat tak kontejner. [5]

2.2 Kontejnerové platformy

Kontejnerové platformy jsou softwarová řešení, která umožňují správu aplikací běžících v kontejneru. Nástroje automatizují jejich vytváření, nasazení, likvidaci a škálování. Tyto platformy se dají rozlišit do několika skupin. [6]

2.2.1 Container engine

Container engine spouští izolované instance kontejnerů na stejném jádře operačního systému, na stejném fyzickém hostiteli. Nejpoužívanější platformou je open source kontejnerizační systém Docker. Známé platformy jsou ale také CoreOS rkt, runC, Hyper-V and Windows Containers nebo LXC. Klíčovou vlastností je kontejner runtime, který komunikuje s jádrem operačního systému za účelem provedení procesu kontejnerizace a konfigurace přístupových a bezpečnostních zásad. V dřívějších dobách měla většina container engine řešení různé formáty kontejner image. Dnes již ale většina používá formát OCI (Open Container Initiative), který specifikuje metadata a vrstvy kontejneru. Formát definuje image kontejneru sestávajícího ze souboru tar pro každou vrstvu a souboru manifest.json, který obsahuje metadata. [7]

2.2.2 Container orchestrators

Orchestrátory kontejnerů automatizují nasazení, správu, škálování, load balancing, síťová propojení, distribuci nových verzí a zotavení při pádu instance. Ulehčují tak práci a správu kontejnerů po dobu jejich života. Jsou vhodné pro instalace se stovkami kontejnerů a více hostitelů. Největším rozdílem je podpora škálovatelnosti, tedy podpora více hostitelů (škálovatelnost aplikace napříč cloudy a datovými centry) s jistotou běhu aplikace stejným způsobem na všech uzlech (hostitelích). Populárními představiteli této technologie jsou bezpochyby Kubernetes, Docker Swarm a Apache Mesos. [8]

2.2.3 Managed container platforms

Jedná se o rozšíření kontejnerového orchestrátoru o další služby, jako je správa orchestrátoru a základních hardwarových zdrojů nebo monitoring. Tuto službu nejčastěji nabízejí velké cloudové společnosti nasazené na jejich cloudech, jako například Google nebo Amazon a další. Jedněmi z představitelů této služby jsou například Google Kubernetes Engine nebo Amazon EKS. [6]

2.3 Monitorování a metriky

Metriky a jejich monitorování mohou prozradit mnohé důležité informace o chodu aplikace, a proto je na místě je sledovat. Metriky nám totiž mohou v mnoha ohledech napovědět, jestli aplikace funguje správně nebo jí docházejí například prostředky a je zapotřebí je navýšit či nastala chyba, kterou je nutné vyřešit. Sledování průběhu metrik v čase nám také může detekovat změnu ve výkonu a tak další analýzou i událost, která k ní vedla. Na základě tohoto zjištění lze predikovat budoucí změny ve výkonu. [16]

Monitorování je využitelné nejen v již produkčních aplikacích, ale může pomoci i při vývoji sledování, jaký vliv bude mít nově nasazená změna na náš systém, nebo odhalování existujících chyb. [16]

Monitorování je proces shromažďovaní metrik, agregace a analyzování hodnot pro zlepšení přehledu jak se daná komponenta/aplikace chová. Monitorovací systém shromažďuje data z různých částí systému a následně je uchovává po určitou dobu za účelem agregace, následné vizualizace v čitelné podobě nebo spuštění automatické reakce, jako je Slack notifikace. [16]

Monitorovací systém se tak dá pomyslně rozdělit na čtyři hlavní komponenty a to shromažďování metrik, která pravidelně čte hodnoty sledovaného systému, uchovávání metrik s časovou značkou a v neposlední řadě jejich vizualizaci nejčastěji v podobě grafů. Poslední hlavní funkcí, kterou monitorovací systémy poskytují, je také aktivní sledování zvolených metrik a v případě změny hodnot podání výstrahy uživateli. [16]

Metriky můžeme dělit dle několika možností. Jednou z těchto možností je podle výstupu. Výstup může být v podobě čítače, stavu nebo běžné hodnoty. Čítač se vyznačuje hodnotou, která může pouze růst (kumulativní). Využívá se například k zachycení počtu chybových stavů nebo doby strávené v user/kernel prostoru. Stav je vyjádřen jako aktivní či neaktivní, tedy nabývá pouze dvou možností. Vystup, jako běžná hodnota, vyjadřuje aktuální stav běžícího procesu, může tedy růst nebo klesat. Další z možností je dělení dle komponenty, na které metriky sledujeme. Příkladem těchto komponent může být procesor, paměť síťová karta, a podobně. [16]

2.3.1 Základní sledovatelné metriky běžících procesů

- **CPU kernel space** — Nebo také system CPU time, který udává čas strávený v operačním systému plněním úkolů na jiném programu.

- **CPU user space** — Udává čas stráveny prováděním programu samotného.

- **CPU usage** — Celkové využití CPU je součet doby v kernel a user prostoru.

- **Throttle** — Pokud je nastaven limit, po jeho dosažení bude aplikace omezena. Výstupem může být počet omezení od spuštění aplikace nebo dobu, po kterou byl program omezen.
- **Celková paměť** — Jedná se o součet RSS a Cache programu
- **RSS** — Necashovatelná paměť, která neodpovídá žádným záznamům na disku. Jedná se například o zásobník a haldu.
- **využití swap** — Swap se označuje jako místo na disku určené pro odkládání záznamů. Využívá se, pokud dochází paměť RAM a je potřeba ji navýšit. Využití swap nám poté udává hodnotu, kolik náš program zabírá místa na disku.
- **Cache** — Množství paměti využívané procesem, ke kterému existuje blok v úložišti. Například při čtení a zápisu na disk se množství cache zvětšuje.

Dále mezi základní metriky můžeme považovat počet read/write operací na disk nebo komunikaci po síti s počtem odeslaných a přijatých balíčků. [11]

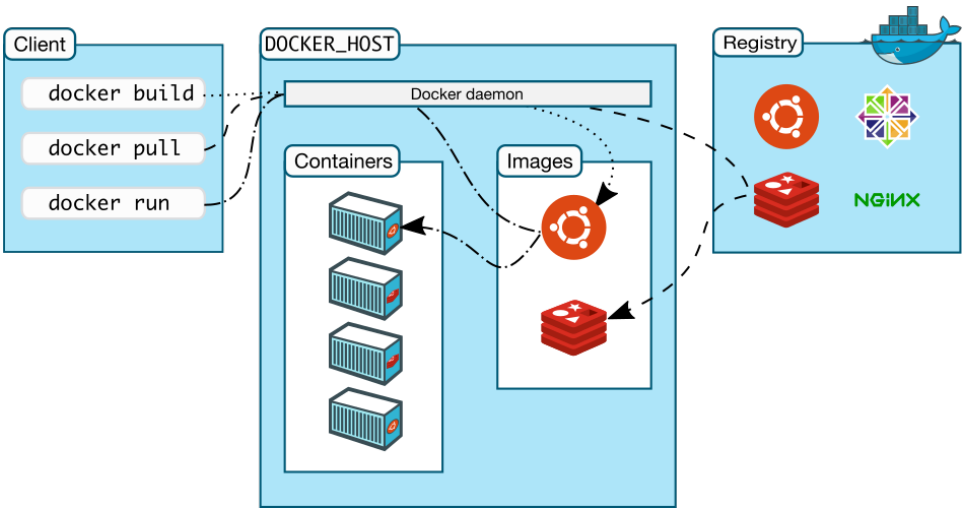
2.4 Docker

Docker je open source platforma pro vytváření, správu a nasazení kontejnerizovaných aplikací. Umožňuje provoz aplikací v izolovaném prostředí v takzvaných kontejnerech, které je možné sdílet s kýmkoli s jistotou stejného chování. Díky tomu je možné kontejner nasadit na cloud nebo do datového centra stejně jako na lokální zařízení. Jedná se jak již bylo řečeno o virtualizaci jádra. [12]

2.4.1 Docker architektura

Docker používá architekturu klient-server. Docker klient komunikuje skrze REST API pomocí socket endpointu nebo network interfasu se serverem, v tomto případě Docker daemon. Pomocí socketu pokud se jedná o komunikaci klienta a daemona na stejném zařízení. Daemon lze také využívat vzdáleně, poté je využit ke komunikaci network interface. Jak architektura Docker vypadá lze vidět na obrázku 2.2. [12]

Daemon se primárně stará o správu objektů jako jsou images, kontejnery, síť a přidělené volumes. Pod správou objektů si můžeme představit veškeré požadavky co objekty vyžadují. Například v případě, kdy kontejner potřebuje přístup k síťovým portům, svazkům úložiště nebo jiným komponentám na úrovni operačního systému, Docker daemon tyto požadavky zajistí. Jedná se o mozek celého systému Docker. Docker daemon vykonává i příkazy přijaté od Docker klienta. Může také komunikovat s jinými Docker daemony. [12]



Obrázek 2.2: Docker architektura [12]

Jak tedy z předchozího odstavce vyplývá, Docker klient je rozhraní, prostřednictvím kterého uživatelé nejčastěji komunikují s nástrojem Docker. Například pomocí příkazu `Docker run`, klient odešle požadavek na spuštění kontejneru daemonu. [12]

Poslední komponentou architektury je Docker registry. Tato komponenta slouží jako prostor pro ukládání bezstavových Docker images. Jedná se o open-source registr, který následně běží na uživatelském zařízení, a kde může uživatel ukládat své image. Společnost Docker rovněž vystavuje svůj vlastní registr s images, které jsou připravené k využít, na Docker Hubu. [12, 13]

2.4.1.1 Docker kontejner

Docker kontejner je spustitelná instance image. Kontejner je v defaultním nastavení relativně izolován a pro jeho propojení se světem lze konfigurovat síť nebo úložiště. Kontejnery v zásadě nejsou perzistentní. Po jejich ukončení jsou všechna data odstraněna. Pro uložení stavů perzistentně je možné využít Docker volume, který je namapován na souborový systém hostitele. Je také možné využít vzdáleného hosta nebo cloudového poskytovatele pro uložení tohoto volume. [12, 14]

2.4.1.2 Docker image

Jedná se o sadu instrukcí pro vytvoření Docker kontejneru. Image jsou často vytvořené z jiných image (mají nějakého předka). Například můžeme mít základní image Alpine rozšířený o naši mikroslužbu. Image můžeme sestavit vlastní anebo využít Docker Hub, kde se nacházejí již předpřipravené image. Pro vytvoření vlastního image se používá soubor Dockerfile. [12]

2.4.1.3 Dockerfile

Dockerfile je textový soubor obsahující instrukce pro sestavení a spuštění Docker image. Každá instrukce (řádek v souboru Dockerfile) je vrstva ve výsledném Docker image. Díky tomuto přístupu lze kompilovat jen ty vrstvy, ve kterých nastala změna. Není tak nutné rebuildovat celá image. Jednou ze základních instrukcí, co tento Dockerfile musí obsahovat, je specifikace image. Ten udává, z jakého image se vychází, například `FROM Alpine`. V případě prázdného souborového systému se začíná `FROM scratch`. Dále můžeme specifikovat instrukce pro přidání souboru, instalaci knihoven, konfiguraci, spouštění skryptů atd. [12]

Dockerfile také může být postaven jako multi-stage build, kdy specifikujeme nejprve první image, kde se provede například build aplikace a výsledek se přesune do finálního image, který již obsahuje jen prostředky nutné k provozu. Zajistí se tím malá velikost finálního kontejneru. [15]

2.4.1.4 Docker Compose

Compose je nástroj pro konfiguraci a spouštění více kontejnerů najednou. Stejně jako u konfigurace Docker image skrze soubor Dockerfile, se Compose konfiguruje pomocí souboru nesoucího název docker-compose.yaml. Konfigurovat se dá síť pro všechny kontejnery, mapování portu nebo připojování svazků a mnoho dalšího. [17]

V docker-compose souboru definujeme služby s cestou k námi vytvořenému Dockerfile nebo k image z Docker hubu, ze kterého následně vznikne kontejner. Po vytvoření a nakonfigurování je možné spustit všechny služby najednou. [17]

2.4.2 Možnosti sledování metrik

Pro sledování metrik existují čtyři možnosti:

- **Sledování aplikace přímo v kontejneru:** Způsob sledování aplikace přímo v kontejneru se liší od použité platformy, v nichž je aplikace nasazena. Například Java aplikace se spouští v enginu JVM. Tento engine poskytuje v základu utilitu jstat, která dokáže uživateli zobrazit statistiky běžící aplikace jako garbage collection, compilation activities a další. Dokážeme tak získat hodně detailní statistiky a metriky běžícího procesu relevantní k použité platformě, které se jinou metodou získat nedají. Tato metoda získávání metrik, jak již postup získání napovídá je velice obtížná a vyžaduje specifickou implementaci pro každou platformu, ale dokáže tak získat metriky, jenž jiné způsoby nedokáží. [19]

- **Sledování pomocí Docker stats:** Příkaz Docker stats lze provést na Docker klientu, jehož výsledkem je živý stream metrik všech běžících kontejnerů na hostitelském zařízení. Příkaz podporuje zobrazení využití

CPU, paměti, limity paměti, bloky přijaté a odeslané na síti a IO operace s diskem. Výsledná data ale nejsou nijak detailní. [11]

- **Sledování pomocí Docker API:** Docker vystavuje API, se kterým lze komunikovat pomocí http protokolu skrze socket, který se nachází na `/var/run/docker.sock` v případe unixových operačních systému. Na tomto socketu poslouchá již zmíněný Docker daemon. Daemon povolí přístup pouze lokálnímu připojení root uživatele. Skrze tento socket lze ovládat prakticky celý docker, jedním z příkazů, který lze přes socket odeslat je požadavek na výpis metrik kontejneru (`GET /containers/$CONTAINER_ID/stats`). Tento příkaz vrátí live stream metriky jako odpověď ve formátu Json. Tato Json struktura obsahuje následně mnohem detailnější metriky, než vypiš Docker stats. Před vykonáním příkazu na zjištění aktuálních metrik kontejneru, je zapotřebí nejprve zjistit `CONTAINER_ID`. Toto id je unikátní pro každý kontejner a dá se vyčíst z detailu listu kontejneru pomocí příkazu `GET /containers/json`. Pro sledování N kontejnerů tedy potřebujeme držet otevřených N spojení, streamující metriky konkrétního kontejneru, s Docker API. [18]

- **Sledování pomocí control groups:** Control groups (cgroups) jsou mechanizmy poskytované linuxovým kernelem, které umožňují správu a alokaci zdrojů procesů a jejich potomků, a mimo jiné umí zobrazit i jejích metriky. Tyto procesy jsou spravovány v rámci skupin. K Cgroups lze přistupovat skrze API v podobě filesystému s pseudo-soubory. Filesystém je umístěn v souborové struktuře `/sys/fs/cgroup` nebo `/cgroup`. V této struktuře je několik podadresářů, kde každý podadresář ve skutečnosti odpovídá jiné hierarchii cgroup. [20] Linuxový kernel podporuje dvanáct control group subsystémů, nás ale zajímají v případě monitoringu pouze tyto:

  - cpuacct - využití procesoru
  - io - limity a využití IO
  - memory - limity a využití paměti

  Jelikož linuxové kontejnery spoléhají na Control groups, lze tedy jejich metriky skrze Cgroups monitorovat. Pro získání například akumulovaného času procesu stráveného v kernelu/uživatelském prostoru postačuje vypsat pseudo-soubor: `cat /sys/fs/cgroup/cpuacct/docker/$CONTAINER_ID/cpuacct.stat` . Jak je vidět z cesty, potřebujeme stejně jako u Docker API `container_ID`. Zmíněné id najdeme dotazem na list kontejneru na Docker API 2.4.2. Obdobným způsobem lze přečíst i ostatní metriky kontejneru.

  Přečtení pseoudo-souboru s příslušnými metrikami je možné bez práv root uživatele, současně se jedná o nejrychlejší a nejlehčí cestu k získání

metrik. Což v případě sledování spousty kontejnerů má zásadní roli, jelikož zaznamenávání metrik se periodicky opakuje v průběhu několika sekund / minut. Tento přístup pro získání metrik nám poskytne stejně podrobné informace jako s využitím Docker API. [21]

Díky všem zmíněným výhodám, které tento způsob monitoringu přináší, je sledování pomocí Control groups nejvhodnější metoda pro tuto práci.

2.5 Docker Swarm

Docker Swarm je open-source orchestrační nástroj, který umožňuje spravovat kontejnery napříč více hostitelskými zařízeními. Docker Engine je rozšířen o swarmkit, který je vrstvou nad Docker Enginem, která implementuje orchestraci. Docker Swarm tedy přímo používá samotný Docker. Jakákoliv služba, která běží na Dockeru lze stejně dobře spustit ve Swarm módu. Díky swarmu se tak docílí vysoká dostupnost služby. Swarm funguje podobně jako Docker Compose s rozdílem provozu servis na více hostitelích. [22]

Swarm používá několik hostitelů s běžící Docker platformou, zvaný nody, přepnutými do Swarm módu a řízeny Swarm managerem. Node se může chovat jako manager, worker nebo vykonává obě role zároveň. [22]

Na začátku před spuštěním swarmu se stanoví počet replik, síť a úložiště společně s otevřenými porty. Docker se následně snaží tento stav udržet. Pokud tedy vypadne nějaký worker node, jsou jeho servisy, které vykonával, přemístěny na jiný node pro dodržení dostupnosti. [22]

2.5.1 Manager node

Pro spuštění servisy podáme požadavek na spuštění manager nodu. Manager následně rozdělí práci workr nodům. Součástí povinnosti manager nodu je i samotná orchestrace, řízení clustru, udržování stavu swarmu a zpřístupnění Swarm HTTP API endpointu. Pro odstranění tzv. single point of failure v podobě jednoho manager nodu je možné zvolit vice managerů. Docker ale doporučuje vždy lichý počet. Platí pravidlo tolerance výpadku (N-1)/2 manager nodů. Po tomto překročení zůstanou servisy sice stále běžet, ale je nutné vytvořit nový cluster pro obnovení. [22]

2.5.2 Worker node

Worker node, čeká na přidělení práce od managera, kterou následně vykonává. V defaultním stavu je každý manager zároveň i worker. Worker se dá příkazem promote povýšit na managera a obráceně v případě potřeby. Na každém worker nodu je spuštěn agent, který podává informace o stavu zadaných úkolů managerovi. Díky tomu má manager informace o všech prováděných úkolech a jejich stavu. [22]

2.5.3 Rozdíly v monitoringu

Monitorování lze provést obdobně jako u samotného Dockeru. Tím, že v podstatě používáme stále stejnou platformu Docker, můžeme použít všechny zmíněné varianty monitorování, dostupné viz 2.4.2. Pokud budeme postupovat naprosto obdobně, zjistíme pouze metriky kontejnerů, které běží právě na klientovi, na kterém provádíme příkaz (docker stats, dotaz na api nebo čtení Cgroups). Proto je nutné monitorování provést/nasadit na všech nodech, které jsou zapojené do Swarmu. Po tomto kroku budeme mít již přehled o všech spuštěných kontejnerech.

Kapitola 3

Analýza

Kapitola Analýza nejprve seznamuje s procesy souvisejícími s monitorováním kontejnerů a následně představuje požadavky na samotnou aplikaci. Na závěr jsou zkoumány již existující řešení v oblasti monitorování kontejnerů.

3.1 Procesy

3.1.1 Sběr metrik

Tento proces se skládá s dílčích procesů, jejichž komunikace a fungování je znázorněno na obrázku: 3.1

3.1.1.1 Identifikace kontejnerů

Jedná se o proces, který se vykonává automaticky a je potřeba ho periodicky opakovat. V případě, kdy se v průběhu monitoringu spustí nový kontejner, je nezbytné ho identifikovat a zaznamenat pro jeho následné monitorování. Součástí identifikace je získání unikátního ID, které je následně použito při čtení metrik tohoto kontejneru.

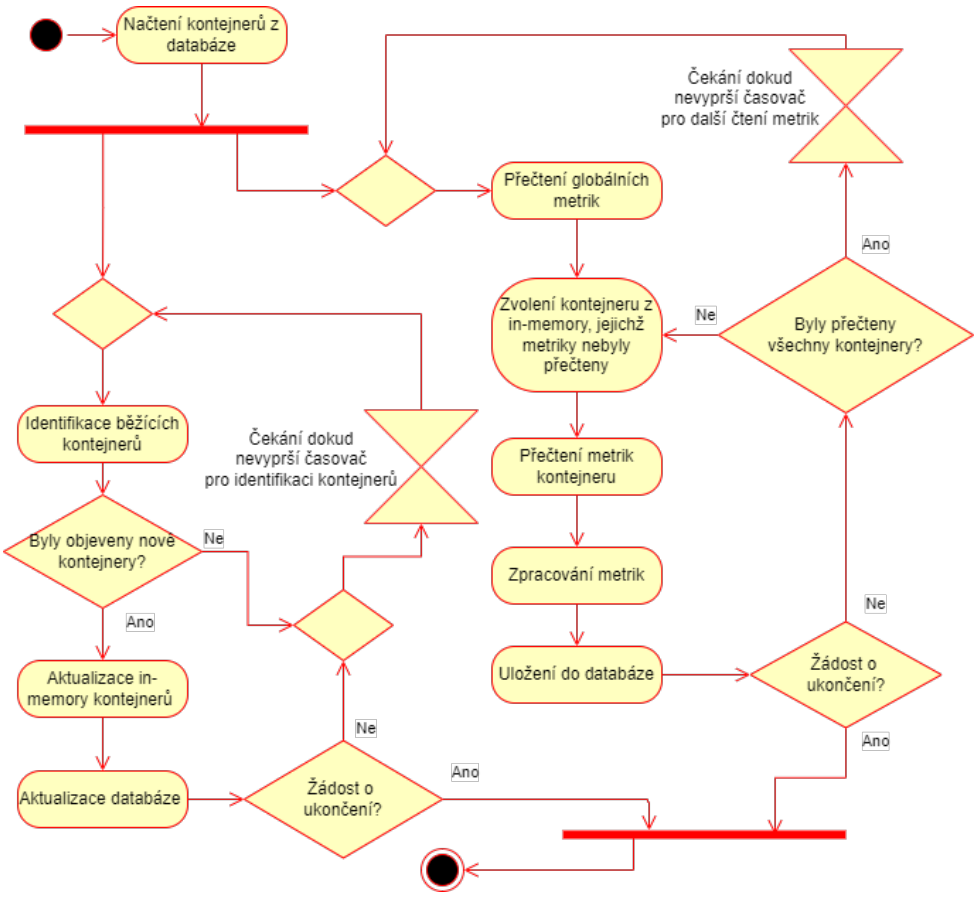
3.1.1.2 Shromáždění metrik

Jde o další proces, který se vykonává periodicky s rozdílem vyšší frekvence čtení oproti identifikování kontejnerů. Postupně se projdou všechny pseudo-subory kontejneru a přečtou se jejich metriky.

3.1.1.3 Zpracování metrik

Přečtené metriky je potřeba před vlastním uložením zpracovat (upravit). Většina metrik je uvedena pomocí akumulované hodnoty, která je pro uživatele

Obrázek 3.1: Proces sběru metrik



hůře čitelná, jelikož postupem času její velikost narůstá. Proto je z dat vypočtena delta, tedy rozdíl dvou čtení za sebou. Tato hodnota je již lépe zobrazitelná v grafech. Další možnou úpravou je procentuální konverze.

3.1.1.4 Uložení metrik

Zpracované metriky je potřeba ukládat pro jejich persistenci. Z důvodu následných analýz je zapotřebí upravené metriky opatřit datovými značkami, to znamená, že společně s naměřenými a následně upravenými hodnotami je nutné tyto doplnit o datum a čas, kdy k měření došlo.

3.1.2 Kontrola pravidel pro notifikaci

Jedná se o periodicky se opakující proces, ve kterém jsou vyhodnocovány zaznamenané údaje. Vyhodnocení spočívá v porovnání zaznamenaných údajů s údaji nastavenými v předem definovaných pravidlech od uživatele. Pravidlo

se vztahuje na kontejner a metriku, která se monitoruje. V případě, že metrika přesáhne určitou stanovenou hodnotu je vytvořena notifikace.

3.1.3 Notifikace uživatele

V případě procesu notifikace uživatele je vytvořená zpráva s podrobným souhrnem společně s informací k jaké události (k jakému porušení pravidel) došlo. Tato zpráva je následně odeslána na uživatelem definovaný Slack server pomocí Incoming Webhooks a zobrazí se jako nová zpráva.

3.1.4 Vyhledání metrik v čase

Metriky jsou, jak již bylo zmíněno výše, vždy ukládány s časovou značkou, dle které je lze vyhledávat. Výsledkem takového vyhledání jsou zpracované metriky ve specifickém časovém úseku. Specifický časový úsek má dvě varianty. První varianta znamená zobrazení aktuálních hodnot (live), druhá varianta spočívá v definování historického data (času) od kterého jsou hodnoty zobrazeny.

3.1.5 Konfigurace pravidel notifikace

Při vytváření notifikací je uživateli umožněno vybrat ze dvou modů a to Threshold a Change, které mají každý svou konfiguraci. Společnými prvky konfigurace pro oba mody je kontejner, který se sleduje, danou metriku, hodnotu a volitelnou poznámku, která se zobrazí v případě uskutečnění notifikace. Poslední společnou hodnotou je časové zpoždění, po kterém se bude znova notifikace opakovat od poslední proběhlé notifikace. Po tento čas se toto pravidlo přehlíží.

- Treshold mód se dá také označit za mód prahové hranice.Tento mód spočívá v porovnání dvou hodnot. První hodnota je přímo zadaná uživatelem, druhá hodnota je vypočtena jako statistická funkce vybrané metriky nad uživatelem definovaným časovým úsekem. Tento časový úsek je pro každou notifikaci individuální, Povolené statistické funkce jsou průměr, minimum, maximum, medián. Vlastní porovnání může nabývat významu jak menší tak větší.

- Change mód můžeme rovněž označit za mód změny. V tomto případě uživatel definuje dva časové úseky. Časové úseky na sebe navazují a jejich délka může být rozdílná. V těchto časových úsecích se vypočte uživatelem definovaná statistická hodnota obdobně jako v módu Treshold. Takto vypočtené hodnoty se porovnají a vznikne nová hodnota - hodnota změny. Takto vypočtená hodnota změny je v posledním kroku porovnána, je-li větší (případně menší) než uživatelem zvolená hodnota změny.

3.1.6 Mazání zastaralých metrik

K procesu mazání zastaralých metrik dochází na pokyn uživatele, kdy již není nutné uchovávat starší naměřené metriky. Mazání probíhá na základě zvoleného kalendářního data, do kterého budou všechny metriky vymazány.

3.2 Požadavky

Hlavním požadavkem je snadná manipulace a nasazení celé aplikace na zařízení. Dále také možnost přistupovat k metrikám vzdáleně pomocí webového rozhraní. Klíčovým faktorem je srozumitelné zobrazení metrik kontejnerů tak, aby byly uživateli přínosem.

3.2.1 Funkční požadavky

F1. Čtení

- F1.1. Čtení kontejnerů
- F1.2. Čtení základních metrik CPU, paměti, disku a sítě

F2. Uchovávaní metrik

- F2.1. Existence více kontejnerů
- F2.2. Časová značka u každého měření
- F2.3. Uchovávání základní metriky CPU, paměti, disku a sítě

F3. Kontrola metrik

F4. Mazání

- F4.1. Mazání kontejnerů
- F4.2. Mazání metrik

F5. Vyhledání

- F5.1. Vyhledání historických metrik
- F5.2. zobrazení live metrik
- F5.3. Zobrazení kontejneru
- F5.4. Zobrazení notifikačních pravidel

F6. Bezpečný přístup

- F6.1. Přihlášení a odhlášení
- F6.2. Změna přihlašovacího jména a hesla

F7. Notifikace

F7.1. Vytvoření nové notifikace

F7.2. Smazání notifikace

F7.3. Modifikace notifikace

F7.4. Nastavení cíle notifikace

F7.5. Provedení notifikace

F8. Konfigurace

F8.1. Nastavení vzorkovací frekvence metrik

F8.2. Nastavení vzorkovací frekvence identifikace kontejnerů

F8.3. Ban list s kontejnery, co se nesledují

3.2.2 Nefunkční požadavky

N1. *Přístup prostřednictvím webového rozhraní* — Aplikaci bude možné obsluhovat prostřednictvím webového prohlížeče s responzivním rozhraním pro podporu různých rozlišení obrazovek. V případě použití příliš malé obrazovky může být snadnost používání aplikace negativně ovlivněna. Samozřejmostí je podpora všeobecně používaných prohlížečů jako jsou Google Chrome, Firefox a Microsoft Edge.

N2. *Rozšiřitelnost* — Systém bude navržen tak, aby jej bylo možné rozšířit o nové funkcionality, případně o podporu dalších kontejnerizačních nástrojů nebo další možnosti integrace s cílovými destinacemi notifikací.

N3. *Nasaditelnost v kontejneru* — Každá část systému bude běžet samostatně v izolovaném kontejneru a to tak, aby výsledná velikost image byla co nejmenší. Dohromady budou části spustitelné automaticky pomocí Docker Compose s nakonfigurovanými prostředky pro jejich běh.

N4. *Bezpečnost* — Pro přístup do aplikace je striktně vyžadováno zadání uživatelského jména a hesla. Při prvotní instalaci aplikace by mohlo být inicializační jméno a heslo vyžadováno jako vstup od uživatele. Tato varianta by ovšem měla vliv na jednoduchost instalace o kterou usilujeme. Z tohoto důvodu byla zvolena varianta automatického nastavení výchozího jména a hesla, které si uživatel může následně změnit.

3.3 Existující řešení

Pro monitorování Docker kontejnerů existuje celá řada jak placených tak i bezplatných platforem. Hledáme-li ovšem celistvá bezplatná řešení, je jich v porovnání s placenými opravdu málo. Je velice obtížné zmínit všechny řešení zabývající se monitorováním, a proto jsou vybrány pouze ty známé, nebo více odpovídající cílům této práce.

Níže se budeme zabývat některými řešeními, přičemž samostatně každé řešení nemusí obsahovat celý systém pro monitorování, tak jak byl nastíněn, ale může se jednat jen o jednu, případně několik z jeho částí. Kombinací těchto řešení pak můžeme docílit využitelného systému.

3.3.1 Bezplatná řešení

Následně si představíme několik aplikací, které můžeme označit jako bezplatná řešení. Budou to CAdvisor, Prometheus, Grafana a Sensu Go. Můžeme také konstatovat, že první tři aplikace se často používají společně při monitorovaní kontejnerů a to z důvodu, že každý z nich je vhodný na jinou oblast při monitorování a z tohoto důvodu se vzájemně se doplňují. CAdvisor je vhodný na sběr metrik kontejneru, Prometheus pro jejich uskladněni a notifikace a Grafana pro přehlednou vizualizaci. Mnohá řešení používají tyto aplikace jako základ, který následně rozšiřují. Takovým řešení je například MetricFire.

3.3.1.1 cAdvisor

cAdvisor je open-source bezplatný nástroj pro monitorování kontejnerů od společnosti Google. Nástroj podporuje grafickou vizualizaci ve webovém prostředí a zobrazování základních metrik CPU, paměti a vstupů/výstupů. Je možné jej integrovat s ostatními prostředími jako Prometheus, Grafana nebo InfluxDB, často se tak i děje. Podporuje sledování jak Docker tak i Kubernetes kontejnerů. [23]

Jeho nevýhodou je chybějící funkce pro upozornění na překročení zvolených limitů metrik. Další nevýhodou je nemožnost sledovat detailní historie dat. Často se proto kombinuje s dalšími prostředími a využívá se pouze jako nástroj pro sběr metrik z kontejnerů. [23]

3.3.1.2 Prometheus

Prometheus je jedním z dalších open-source bezplatným řešením pro monitorování metrik kontejnerů od společnosti SoundCloud. Na rozdíl od cAdvisor řešení podporuje i upozorněni. Prometheus poskytuje zaznamenávání, ukládání a agregaci metrik. Pro zaznamenávání metrik je nutné, aby klient vystavoval HTTP endpoint `/metrics` , který vrátí seznam všech metrik a jejich hodnot. Docker podporuje tento HTTP endpoint a lze ho nakonfigurovat pro potřeby monitorujícího nástroje Prometheus, nebo je možné využít jiného řešení pro čtení těchto metrik. [24]

Pro grafickou vizualizaci je Prometheus závislý na dalším softwaru a to je Grafana, která se používá pro vizualizaci metrik v podobě přehledných grafů. Bez tohoto rozšíření lze pro vizualizaci použit jedno ze dvou řešení. Expression browser je jedním z nich, které podporuje zadání vlastního výrazu, jehož výsledkem je tabulka nebo graf. Poslední možností je Console teplates, kdy pomocí jazyka Go teplating language jsem schopni vytvořit vlastní konzoli. Ve

většině případů se ale nejedná o typické použití a dává se přednost přehlednější vizualizace pomocí Grafany. [24]

Velkou nevýhodou stojící za zmínku je vysoká složitost nasazení a konfiguraci. Například nastavení upozornění je možné pouze prostřednictvím souborů s vlastní syntaxí. Tyto nesnáze často zapříčiní nevyužití tohoto řešení. [24]

3.3.1.3 Grafana

Grafana je open-soruce řešení pro analýzu a vizualizaci dat v podobě webové aplikace vyvinuté společností Grafana Labs. Převážně se používá na vizualizaci metrik. Umožňuje různorodé možnosti vizualizace a také vytvoření vlastního dashboardu se zvolenými grafy a tabulkami. Je dobře přizpůsobitelná dle potřeb uživatele. Grafana podporuje širokou škálu zdrojů dat, od databází jako InfluxDB, MySQL, PostresSQL a dalších. Rovněž podporuje integraci s monitorovací platformou Prometheus, se kterou je nejvíce využívána. [25] Náhled na dashboard rozhraní řešení Grafana: 3.2

Obrázek 3.2: Dashboard nástroje Grafana [26]

3.3.1.4 Sensu Go

Sensu je další monitorovací nástroj uvedený jako open-source s bezplatným přístupem v omezené podobně. Tento projekt vychází z řešení Sensu monitor, který je již označen jako EOL (End of life) ale je stále přístupný skrze github. Sensu go je možné používat bezplatně s omezením na 100 nodů. Pro větší počet cloud nodů je nutné již rozšíření, které je zpoplatněno. [27] Jedná se o cloudové řešení pro monitorování a notifikace stavu služeb a telemetrických dat. Podporuje nástroje jako Grafana pro zobrazení dat ale rovněž integruje

Ansible, EC2, InfluxDB, PagerDuty, Puppet, Rundeck, Saltstack, Slack nebo Sumo Logic. Pro monitorování nasazuje na každý nodu Seunsu agenta, který odesílá data do Sensu-backend aplikace. Jeho instalace a nastavení je v porovnání s Prometheuse monitorovacím řešením jednodušší ale stále náročné. Je zaměřenější více na monitorování sítě a života aplikací, než jejich metrik jako je CPU nebo paměť, Sensu agent lze ale nakonfigurovat tak, aby tyto metriky také sledoval. [27]

3.3.2 Placená řešení

Ve všech dalších zmíněných řešeních se jedná o plné aplikace, které obsahují veškeré části potřebné pro monitorování a mohou tak fungovat samostatně. Jedná se o placená řešení, u kterých je většinou možné si vyzkoušet jejich plnou verzi na několik dní zdarma.

Existují i další zpoplatněná řešení, která nebudou již rozebrány detailně, jelikož poskytují téměř totožné funkcionality, jako programy pro monitorování, které budou zmíněné v další sekci. K těmto programům patří mimo jiné populární řešení SolarWinds Server & Application Monitor, případně řešení AppOptics Docker Monitoring nebo Paessler PRTG, které jsou přímo zaměřené na sledování kontejnerů běžících na platformě Docker.

3.3.2.1 DataDog

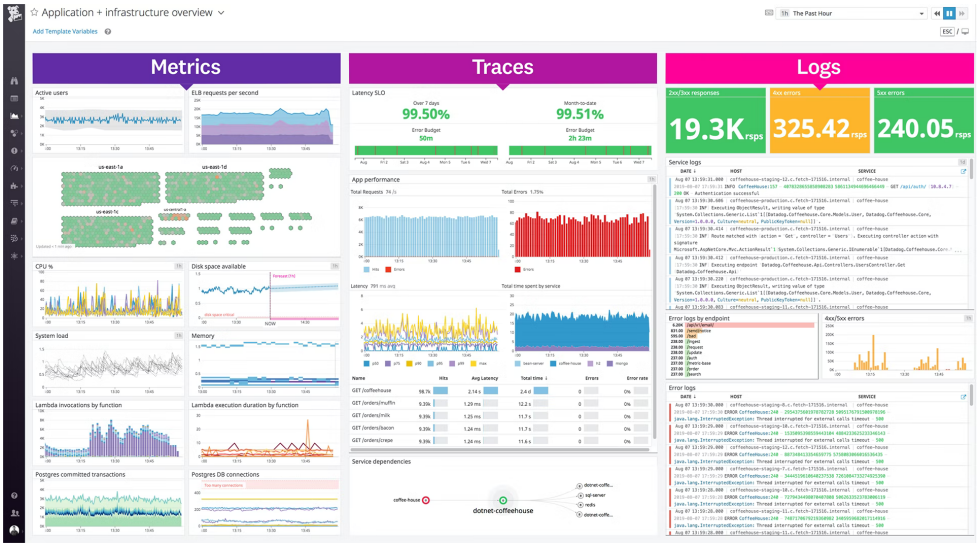
DataDog je cloudové řešení pro monitorování a notifikace. Podporuje nejen sledování kontejnerů ale i samotných aplikací, které v kontejnerech neběží. Lze využít také pro monitorování sítě nebo infrastruktury. Podporuje více než 500 integrací produktu od Amazonu až po Microsoft. Zahrnuje tak širokou škálu nejpoužívanějších platforem. V případě neexistující integrace je umožněno vytvořit vlastní prostřednictvím DataDog API. DataDog podporuje jak standardní metriky, mezi které patří využití procesoru, paměti a sítě, tak podporuje i detailní metriky specifické k dané platformě. Tato rozsáhlá podpora metrik je hlavně díky jeho mnohým integracím. [23, 28]

Jedná se o kompletní řešení od získání metrik, uložení až po vizualizaci a notifikace. Umožňuje návrh vlastního dashboardu pomocí drag-and-drop způsobu rozložení. Uživatel si tak může uspořádat a zobrazit potřebné informace o běžící aplikaci dle své potřeby. Vše od nastavení notifikací až po vizualizaci dat je možné skrze webové rozhraní. Dashboard tohoto řešení je vidět na obrázku 3.3. [28]

Notifikace jsou robustní a propracované oproti ostatním řešením, které podporují upozorňování. Umožňují sledovat jak změnu, tak i hraniční hodnotu s detailním nastavením. Způsob notifikace v této práci je inspirován tímto řešením. [28]

Přes veškeré zmíněné výhody se jedná spíše o řešení pro obrovské společnosti, což je dáno mimo jiné též vyššími nároky na jeho velikosti způsobené

jeho širokou integrací. Jedná se o náročný monitorovací systém. Ke zmíněným nevýhodám také patří i vysoká cena tohoto produktu. [23]



Obrázek 3.3: Dashboard nástroje DataDog [29]

3.3.2.2 Sysdig Monitor

Sysdig poskytuje všechny základní požadavky na monitorování kontejnerů, upozorňování a vizualizaci metrik. Nativně podporuje kontejnerizační nástroje jako Docker, Kubernetes nebo Mesos. Podporuje integraci i přímo s některými platformami, ale ne v tak velké míře jako DataDog. Pro sběr metrik používá eBPF (Extended Berkeley Packet Filter) agenta, Kubernetes, Prometheus, integrations nebo cloud services. [30, 31]

Pro jeho fungování vyžaduje instalaci kernel hlaviček do hostitelského operačního systému. Jedná se o technologii jádra, která umožňuje spouštění programů bez nutnosti měnit zdrojový kód jádra nebo přidávat další moduly. [31]

Sysdig umožňuje jak zobrazení historických, tak aktuálních metrik přímo v dashboardu přístupného pomocí webového rozhraní. Skrze toto rozhraní je možné konfigurovat celý systém monitorování a upozorňování. V dashboardu lze srovnávat metriky za určité období a ty následně uložit pro pozdější analýzu. Jednou z výhod je umožnění přístupu s různým oprávněním, které se dá prostřednictvím webového prostředí také spravovat. [30]

Jeho hlavní nevýhoda je nutnost instalace kernel hlaviček společně s ne příliš intuitivním UI. Základní balíček tohoto systému není příliš finančně nákladný, ale tyto náklady velice rychle rostou s rostoucími požadavky. [31]

3.3.2.3 SemaText

Dalším známým monitorovacím řešením v sekci placených možností je SemaText. Tento systém slouží k monitorování výkonu aplikací, správa logů, notifikaci a detekci anomálií. Lze jej nasadit jak na cloud, tak i on-premise. Pro integraci SemaText poskytuje agenty pro několik platforem, přičemž jednou z nich je i kontejnerizační nastroj Docker nebo Kubernetes. Jedná se o nástroj jednoduchý na nasazení a lze jej nasadit i do Docker kontejneru jako velice malou a efektivní aplikaci, která následně monitoruje Docker kontejnery. Využívá též funkce pro identifikování nových kontejnerů za chodu a následného automatického monitorování. [32, 23]

Ovládání SemaText aplikace je možné zcela prostřednictvím webového rozhraní, podporující též vlastní dashboard. Uživatelské rozhraní je možné vidět na obrázku 3.4. Do webového rozhraní je možné přistupovat pomocí konfigurovatelných uživatelských profilů s různým oprávněním. [23, 32]

Nasazení a konfigurace není příliš složitá. Nevýhodou tohoto řešení je chybějící dokumentace pro některé starší agenty a také špatné integrace s bezpečnostními zařízeními. [23]

Jedná se o zpoplatněné řešení s hodinovou sazbou za jednoho hostitele. Uchovává data pouze dva měsíce, pro rozšíření je nutný příplatek, jinak není nijak funkcionálně omezené. [23]

Obrázek 3.4: Dashboard nástroje SemaText [33]

Kapitola 4

Volba technologií

V této kapitole budou rozebrány použitelné platformy pro monitorovací systém navrhovaný v této práci. Monitorovací systém je rozdělen do několika částí/aplikací a to čtení metrik, databáze, kontrola metrik a front-end. Pro každou část bude zvolena nejvhodnější platforma,která bude následně použita při implementaci. Detailněji se kapitola Analýza technologií zaměří na volbu databázového systému, jelikož bude podstatně podílející se částí na zatížení hostitele. Proto bude v rámci této volby databázového systému finální rozhodnutí rozhodnuto na základě vlastních benchmarků dvou zvolených databází.

4.1 Volba platformy pro čtení metrik

4.1.1 Požadavky

Jak již bylo zjištěno z analýzy způsobu monitorovaní kontejnerů viz 2.4.2 bude potřeba číst opakovaně mnoho pseudo-souborů. Je tedy na místě uvažovat o takové platformě, která dokáže rychle číst a pracovat se soubory. Vzhledem k potřebě nasazení servisy pro čtení metrik na každý Docker node, je nezbytné myslet na nízké hardwarové nároky. Potřeba efektivního využívání jak paměti, tak i procesoru je na místě. Naproti tomu je zbytečné uvažovat o jazycích, které k běhu vyžadují interpretr, jelikož programy kompilovány do nativního strojového kódu, bývají rychlejší než interpretovaný kód. Důvodem je, že proces překladu kódu za běhu zvyšuje režii a může způsobit, že program bude celkově pomalejší. Nevýhodou kompilovaných programů je sice vetší velikost a závislost na platformě, ale to nám v tomto případě nevadí. [34] Proto budou brány v úvahu jen jazyky které nepotřebují interpretr.

4.1.2 Technologie

4.1.2.1 Fortran

Fortran je počítačový programovací jazyk z rodiny kompilovaných jazyků známý již řadu let. Zkratka Fortran vychází ze slov FORmula TRANslation (překlad formule). Byl vyvinut společností IBM a jeho první verze vyšla roku 1957 a přinesla revoluci v programovacích jazycích. Do této doby se používal pro programování primárně jazyk asembler známý svou obtížností pro psaní pokročilejších programů. Fortran patří k relativně malým jazykům, který má snadnou učící křivku. I přes jeho velkou popularitu a přicházející nové aktualizace jej dnes do značné míry nahradily jazyky čtvrté a páté generace a jeho popularita tak upadla. Stále má silnou podporu u věděckých komunit, na akademických půdách nebo také v organizacích zabývajících se předpovědi počasí či finančním obchodováním díky jeho přednostem v rychlosti některých typů numerických výpočtů. [36]

Fortran je i po řadě let stále velice rychlý i na dnešní poměry jazyků a je vhodné jej nasadit tam, kde je potřeba řešit výkon. Zejména vyniká při zpracování polí. Jeho rychlost je zřetelná především pokud lze problém popsat pomocí jednoduchých datových struktur a především matic. Výhodou při zrychlování programu je nativní podpora paralelizmus. [36]

4.1.2.2 Go

Go nebo také Golang je open-source programovací jazyk, jehož popularita stále roste. Jedná se o poměrně mladý jazyk, jehož vývoj započal roku 2007 společností Google a následně o dva roky později byl vydán (tedy v roce 2009). Zpočátku se jednalo pouze o experiment, který měl eliminovat známé problémy jiných jazyků, ale v průběhu se stal plnohodnotným a používaným jazykem. Díky jeho uvedení jako open-source k jeho vývoji a zlepšení dovedností přispívají řady velkých společností, ale rovněž rostoucí komunita. [37]

Programovací jazyk Go se primárně používá pro systémové programování a programy související se sítí a infrastrukturou. Jeho cílem bylo nahradit konvenční řešení jako je Java nebo Python. Jedná se o jazyk zaměřený na jednoduchost, spolehlivost a také účinnost. V průběhu nových verzí se jeho syntaxe příliš neměnila a díky jeho důrazu na jednoduchost při návrhu jazyka je relativně snadný na pochopení a práci. Jedná se též o poměrně rychlý jazyk díky jeho implementaci lehkého manažera vláken, který spotřebovává méně prostředků než vlákna v operačních systémech. [37]

Jazyk Go nejvíce vyniká, pokud jde o infrastrukturu. Některé z dnes nejpopulárnějších infrastrukturních nástrojů jsou napsány v jazyce Go, jako jsou Kubernetes, Docker a Prometheus. Velcí giganti jako Google, DropBox, Docker, Uber nebo PayPal založili své produkty na tomto jazyce. Tímto výčtem jeho využití nekonči, dnes jej využívají tisíce společností. Není tak divu, že se stává čím dál více oblíbeným. [37, 36]

4.1.2.3 C/C++

C je procedurální programovací jazyk, jenž začal nabývat popularity v osmdesátých letech. Původně jej vyvinul Dennis Ritchie jako systémový programovací jazyk pro psaní operačních systémů. Oproti assembleru se v době vzniku jednalo o jazyk s relativně jednoduchou sadou klíčových slov pro programování. Mezi jeho hlavní vlastnosti patří nízkoúrovňový přistup k paměti. Udržování paměti je tedy zcela ponecháno na programátorovi, což přináší potencionální nevýhodu v riziku špatné správy, ale také výhodu v podobě naprosté kontroly nad pamětí. Nakládání s pamětí je jedním z důvodu proč je programovací jazyk C tak rychlý a to i na dnešní poměry. Jazyk totiž nemusí řešit starosti vzniklé s udržováním paměti čímž šetří zdroje. [38]

Rozšíření jazyka C přišlo v roce 1985 navržené Bjarne Stroustrup nesoucí název C++. Toto řešení se následně stalo jedním z nejúčinnějších a nejrychlejších vysokoúrovňových jazyků. Ačkoli je dnes již mnohem větší množství jazyků, přes to se z dnešního pohledu c++ řadí k těm nejrychlejším. Je široce používané programátory pro jeho rychlost provádění a standardní knihovny šablon STL. C++ je navíc od C již objektově orientovaný a funkcionální jazyk. Poskytuje jak zpětnou podporu jazyka C, tak přináší nové možnosti se zachováním stále vysoké rychlosti. Jeho výhodou nad ostatním je rychlost a současně vysoká portabilita. Mnoho projektů, včetně kompilátorů, cloudových úložných systémů, databází atd., je stále napsáno v jazyce C++. Jedněmi z nich jsou například velice známe platformy Youtube nebo Spotify. [38]

4.1.3 Shrnutí

Ačkoli je Fotran v některých ohledech dokonce rychlejší než C++, nebyl zvolen jako technologie pro čtení metrik. Jeho přednosti v podobě rychlé práce s poli nepřinášejí dostatečný přínos. Fortnar již není v současné době tak používán a zaostává v nabídce rozšířitelných knihoven, které jsou potřebné pro snazší vývoj monitorovacího systému. Jedná se především o knihovny pro práci s XML dokumenty nebo připojení do databáze či implementace komunikačních protokolů.

Jazyk Go je na druhou stranu mnohem jednoduší a přehlednější. Předchází tak v mnoha ohledech riziku vzniku chyb a zrychluje tím vývoj. Je všeobecně používaný s velkou komunitou a užitečnými knihovnami. Jedná se ve své třídě o velice rychlý jazyk, který se zdá být jako vhodná volba díky jeho přednostem.

S rychlostí, které dosahuje jazyk C++ se ale Go nemůže rovnat. Jazyk C++ je oproti jazyku Go na nižší úrovni a neřeší některé prostředky jako paměť automaticky, ale nechává je na programátorovi. Implementace se tak stává výrazně náročnější a delší. Díky větší blízkosti ke strojovému kódu jsou programy psané v C++ efektivnější a lehčí ve výsledné zkompilované podobě.

I přes riziko vetší pravděpodobnosti vzniku chyb a obecně složitějším implementacím, byl vybrán programovací jazyk C++. A to především díky jeho

skvělým rychlostem, a efektivního využití zdrojů, které budou nezbytné při sbírání metrik.

4.2 Volba databáze

Budeme-li chtít, aby vyvíjená aplikace byla co nejefektivnější, bude na volbě databázového systému záležet pravděpodobně nejvíce.Jedná se o prvek, se kterým většina komponent aplikace komunikuje a zastává pozici prostředníka. Její rychlost na dotazy a zápisy bude bottleneck celé aplikace. Proto je její volba stěžejní.

4.2.1 Požadavky

Při volbě databáze bude potřeba zvážit kompromis nad výkonem a hardwarovými prostředky, které databáze vyžaduje k chodu. Přílišná alokace zdrojů pro databázi by mohla omezit vlastní činnost hostitele.

Databáze bude přijímat nepřetržitě metriky v krátkém časovém rozestupu v řádu sekund a může tak hrozit zahlcení. Dle společnosti zabývající se monitorováním je medián běžících kontejnerů v Dockeru na jednom hostitelském zařízeni přibližně 12. [35] Tento členek vyšel roku 2018 a je pravděpodobné, že toto číslo je již dnes překonané a bude vyšší. Dnes tedy předpokládejme číslo okolo 20 kontejnerů, u kterých neustále po dobu několika sekund čteme jejich metriky a ukládáme do databáze (28 metrik). Zvážíme-li dále použití více hostů, je nápor na databázi ještě větší. Z uvedeného je jasné, že se bude jednat o nápor, který musí databáze zvládnout. Je zapotřebí, aby databáze stihla data uložit dříve než začne nové čtení metrik, jinak dojde k zahlcení.

S potřebou uložit hodně metrik je v kontrastu potřeba malé fyzické stopy databáze. Data musí být uložena optimálně tak, aby nezabírala mnoho místa. Zároveň nechceme využít veškerý výkon hostitelského zařízení jen pro databázi. Je tedy příhodné uvažovat o takovém typu, který bude mít nízké nároky jak na paměť, tak i procesor a bude schopna přijímat hodně požadavků od více hostitelů zároveň.

4.2.2 Typ databáze

Nejprve je potřeba zvážit výběr vhodných typů databází, před konkrétní volbou platformy. Existuje obecně mnoho typů a přístupů dělení databázových systémů pro ukládání různých druhů dat, a proto budou zmíněny pouze některé. Databázové systémy se dají dělit na základě modelu, umístění databáze, designu, hostitele nebo umístění dat. Existují určitě i další kritéria dělení, která nebyla zmíněna. Často ale jako základní dělení je vnímáno rozdělení dle modelu na relační a nerelační nebo také NoSQL databáze jak se obvykle označují. [39]

Konvenčním typem vhodným na všeobecné použití jsou relační databáze, jelikož díky jejich modelu v podobě relací dokáží snadno reflektovat většinu domén. Data jsou uložena do tabulek se sloupci a řádky, kde řádek reprezentuje jeden záznam. Pro komunikaci se běžně používá Structured Query Language (SQL). Relační databáze jsou velice spolehlivé hlavně díky souladu s ACID (Atomicity, Consistency, Isolation, Durability). [39]

V kontrastu se standardními relačními databázemi jsou databáze typu NoSQL. NoSQL databáze se vyznačují svým specifickým použitím pro konkrétní případy. Jedná se o velkou skupinu databází, které se dále dělí na dokumentové, Key-value, grafové column based nebo také time series (TSDB) případně další. [39]

Každý typ ze jmenovaných NoSQL databází má specifický model. V případě Dokumentové databáze jsou data ukládána, jak již název napovídá, v podobě dokumentů.Často se jedná o ukládání v XML struktuře. U Key-valů jsou data ukládána v podobě klíče a příslušné hodnoty s výhodou rychlého vyhledaní podle klíče. Grafové databáze využívají struktury s uzly a hranami, které je spojují. Ze zmíněných typů NoSQL databází se pro uložení kontejnerů a jejich dlouhodobých metrik žádný příliš nehodí. Tento fakt je způsoben primárně odlišným zaměřením jejich domén. Je zde ale jeden typ NoSQL databáze přímo určený pro zaznamenávání hodnot v průběhu času. Jedná se o databáze typu Time series. [39]

Time series datadabáze také označovaná jako TSDB je databáze časových řad určená k ukládání datových bodů, které jsou spojené s časovým razítkem. Časové razítko poskytuje každému datovému bodu kontext v tom, jak společně tyto body souvisí. Je tak možné tyto body analyzovat a vyhodnotit určité závěry. Časové řady jsou často nepřetržitým tokem dat ze senzorů, cen akcií a atd. Pod daty respektive datovými body si můžeme představit jednotlivá měření nebo události, které sledujeme v průběhu času. Time series databáze umožňují ukládání velkého objemu dat s razítky v úsporném formátu, který umožňuje rychlé vkládání a vyhledávaní. Je tedy vhodná i na udržování metrik kontejnerů a jeví se jako obstojný kandidát pro tuto práci. [40]

Další dělení databází z pohledu umístění, designu a hostitele již nebudeme rozebírat, jelikož se nejedná o kategorie nijak zásadní pro potřeby této práce. Databáze bude nasazena do izolovaného prostředí Docker kontejneru a proto není nutné zvažovat cloudová řešení. Distribuované řešení není nijak stěžejní, může sice odlehčit síť a rozprostřít zatížení mezi více hostitelů, ale za následek větší náročnost a celkového zatížení. Proto distribuovaná řešení také nebudeme zvažovat.

Z výše uvedeného vyplývají dva použitelné typy databází, které se jeví jako vhodné pro tuto práci. Jedním je tradiční relační databáze s obecným použitím a druhým typem je databáze typu TSDB pro zaznamenávání hodnot v čase.

4.2.3 Použitelné technologie

4.2.3.1 SQLite

SQLite je velice populární databáze v podobě C knihovny vydané v roce 2000 vývojářem D. Richard Hipp. Nejedná se o samostatnou aplikaci, ale o knihovnu, která databázi implementuje. Díky tomu je databáze velice malá (500KB), rychlá a spolehlivá. SQLite je typu relační serverless databáze. Díky serverless vlastnosti nevyžaduje k běhu žádný server, ale funguje na bázi jednoho datového souboru. Tento soubor se chová jako celkové a jediné úložiště databáze. V případě uložení nebo vyhledání dat se vždy pracuje s tímto souborem. [41]

Její obrovskou výhodou je malá velikost, rychlost a nevyžadování běžícího serveru. Díky čemuž nespotřebovává žádné prostředky v nečinnosti. Neexistence databázového serveru je ale zároveň také nevýhodou, projevující se v časté komunikaci s diskem nebo přístupu více aplikací vyžadujících zápis. Tento stav lze považovat za kritický, jelikož nelze zjistit, zdali některé aplikace zrovna do souboru s databází nevyžadují zápisu. Mohou tak vzniknout chyby v podobě nezapsání nových metrik, nebo k celkovému selhání souboru s databází. Tomuto faktu se dá vyhnout v podobě vytvoření vlastního serveru, který bude požadavky od jednotlivých aplikací řídit.

4.2.3.2 QuestDB

QuestDB je column-oriented open-source databáze typu TSDB vydaná v roce 2016, přímo zaměřená pro zaznamenávání metrik. Je optimalizovaná tak, aby zabírala co nejméně paměti oproti ostatním TSDB databázím. Pro její běh vyžaduje běžící server na pozadí na rozdíl od SQLite. QuestDB databáze je psaná v jazyce Java a C, podporuje dotazování pomocí SQL a implementuje několik protokolů pro komunikaci jako PostgresSQL nebo Influx Line protokol. Lze k ní také přistupovat pomocí REST API. [42]

Výhodou této databáze je specializace na uchovávání metrik a jejích optimalizované ukládání. Další z řady výhod je podpora široké škály protokolů společně s možností integrace softwaru Grafana pro vizualizaci dat.

Nevýhodou jsou chybějící známé databázové funkce jako primární klíče u tabulek nebo vazby mezi nimi. Díky tomu, že se jedná o poměrně nové řešení, není zdaleka vyladěná jako jiné databáze, čemuž nasvědčuje i malé množství návodů nebo podrobnější dokumentace. Vývojáři této databáze, ale přicházejí postupně s novými verzemi a je tak pravděpodobný budoucí pokrok a přidání chybějících funkcionalit.

4.2.3.3 InfluxDB

InfluxDB je jednou z nejoblíbenějších TSDB databází vyvinutou společností InfluxData s vydáním v roce 2013. Databáze je vydaná jako open-source na-

psaná v jazyce Go a navržená tak, aby poskytovala vysoce škálovatelný stroj. Pro komunikaci používá jazyk InfluxQL, jenž je velice podobný dotazovacímu jazyku SQL. Poskytuje velmi efektivní nástroj při shromažďování, ukládání, dotazování, vizualizaci a provádění akcí s proudem přicházejících dat. Obsahuje řadu funkcí pro komplexní ukládání dat s nízkými nároky na velikost disku, nebo automatické mazání zastaralých dat. Podporuje rovněž nástroj pro vizualizaci dat s dashboardem, nebo možnost monitorování a upozorňování. [43]

Jednou z výhod této databáze je schopnost zpracovat miliony dat za pouhou sekundu. Nelze opomenout ani velké množství funkcí. Tyto funkce z InfluxDB nedělají pouhou databázi, ale komplexní monitorovací službu. [44] Jedná se již o náročnější zařízení na provoz a vyžaduje větší nároky na hardware.

4.2.4 Porovnání QuestDB a SQLite

Jako vhodná řešení zmíněného problému byly zvoleny dvě databáze, které je nezbytné detailněji prozkoumat a otestovat v simulovaném prostředí. První zvolenou databází je SQLite kvůli své velikosti a malé náročnosti na provoz. Druhá databáze byla zvolena QuestDB, kvůli zaměření na ukládání metrik a optimalizací jejích uložení s rozumnými nároky na hardware. Cílem tohoto měření je zjistit, která databáze bude pro potřeby této práce vhodnější.

4.2.4.1 Testovací prostředí

Veškeré výpočty byly provedeny na PC: (OS: Ubuntu, CPU: Core i5-4210U). Jako programovací jazyk byl zvolen jazyk C++ a knihovna pro zaznamenávání času chrono. Pro získávání metrik byly využity metric files v cgroup. Jak SQLitre tak QuestDB databáze byly testovány přímo v Docker kontejnerech. SQLite bylo umístěno do `light Ubuntu 20.4` a QuestDB do `debian stretch slim` kontejneru.

4.2.4.2 Postup měření

Měření probíhalo nad obdobným modelem, jako bude použit u finální aplikace. Pro simulaci příkazů insert i select byly použity reálné podoby záznamů, které budou i ve skutečném prostředí.

- Měření select time — Čas proběhnutí select příkazu se měřil nad 2 milionem záznamů s požadavkem na konkrétní data daného kontejneru. Tento proces byl 50x opakován a stanoven průběžný čas provedení tohoto příkazu.

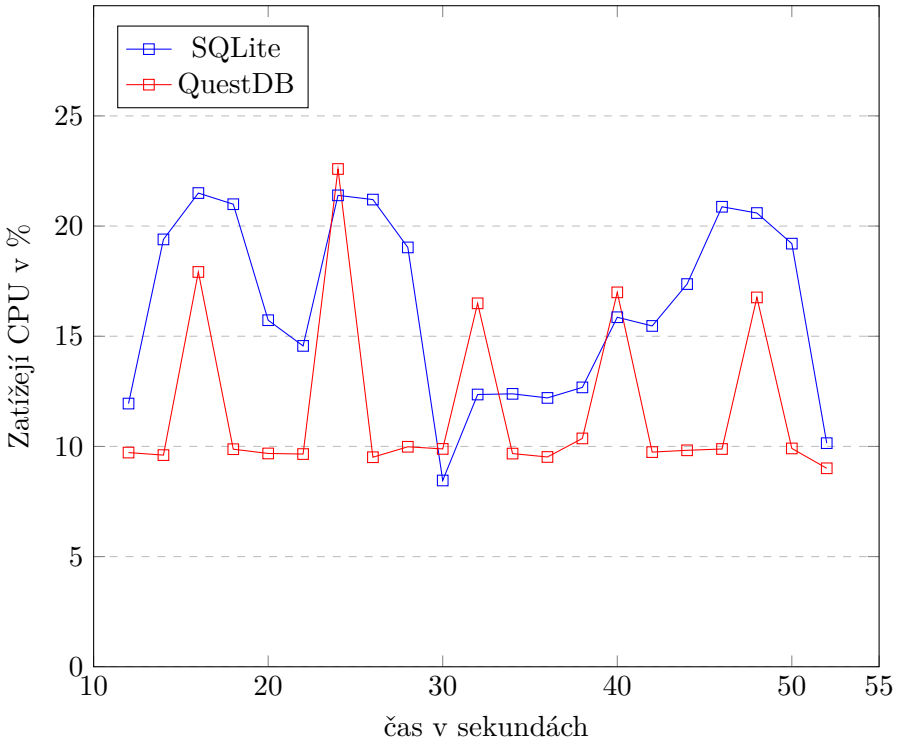
- Měření insert time — Čas vložení dat do databáze byl měřen obdobně jako u select příkazu. Testovalo se vložení 3000 záznamů do databáze s následným průměrem trvání vyhodnocení tohoto příkazu.

- Zatížení serveru — Zatížení bylo měřeno ve stavech idle a load. Zatížení databází bylo provedeno pomocí příkazu insert, kdy se do databáze v průběhu 8 sekund vložilo 3000 záznamů. Tento čas byl změřen na základě horní hranice počtu vložení dat do databáze SQLite. Při jejím překročení již nebyla databáze schopna v časovém úseku data uložit.

4.2.4.3 Naměřené hodnoty

Databáze	insert ms	select ms	CPU % idle	CPU % load	Pamět MB	IO zápis MB
SQLite	9007,04	**215,40**	**0**	17,30	**21,14**	21,14
QuestDB	**38,10**	469,72	9,821	**12,48**	141,38	**0**

Zátěž CPU při opakování příkazy insert po 10 sekundách



4.2.5 Shrnutí

Z dat naměřených v předchozí sekci se dá vyčíst hlavní omezení SQLite a to v podobě času potřebného pro provedení příkazu insert a s tím související zapsání dat do databáze. Tento čas je v případě jednoho kontejneru přibližně 9,4 ms. Jelikož je žádoucí, aby program dokázal zaznamenávat metriky alespoň v intervalu jedné sekundy, dává nám tedy SQLite omezení na provoz přibližně 100 kontejnerů. Po přesáhnutí zmíněné hranice se začne celý program pro záznam metrik zpožďovat a zvětšovat tak interval zaznamenávání. Naproti tomu

databáze QuestDB ani po provedení několika milionů vložení se této hranice nedotkla. Data nejsou okamžitě perzistentně uložena po provedení příkazu insert, jelikož server používá buffer, ale uloží se až po několika sekundách. Což v našem případě použití nebude stěžejním parametrem, jelikož uživatel nepotřebuje vidět metriky okamžitě v řádu sekund, ale za dostatečné se považuje i několikasekundové zpoždění.

Z hlediska select dotazu je pro menší datasety výhodnější použít SQLite a to díky indexaci dle cizího klíče a také rychlému přístupu, kdy k datům rovnou přistupujeme a není nutné je přeposílat prostřednictvím rozhraní, jako je to v případě QuestDB. Při použití složitějších dotazů (vyhledávání dle data s agregací hodnot) se tento rozdíl zmenšuje, dokonce i ,QuestDB získává na rychlosti oproti SQLite. V cílové aplikaci ale rychlosti dotazu select nejsou nijak zásadní, jelikož jsou použity jen ve specifických případech. Těmito případy jsou podklady pro uživatelské vizualizace nebo podklady pro notifikace.U těchto informací v zásadě nepožadujeme okamžitou odezvu případně reakci v řádu jednotek sekund.

Výhoda nulového zatížení zařízení v klidovém stavu u SQLite databáze nepřinesla velkou výhodu. Metriky budou přiházet v takové míře a objemu, že se v takovém stavu databáze ve většině případu nacházet nebude. Jediná možnost klidového stavu je v případě malého množství kontejnerů a velkého časového rozestupu při čtení metrik. A proto tato výhoda SQLite databáze nepřináší v konečném důsledku žádné výhody. Z analýzy předchozího grafu je viditelný rozdíl v zatížení ve prospěch QuestDB. V průměru bylo vyžadováno o 5 % CPU méně oproti SQLite.

Při porovnávání paměťové náročnosti obou databází vyhrála na plné čáře SQLite s neuvěřitelnými 20MB. U QuestDB byly paměťové nároky o něco větší, ale stále velice nízké na poměry běžných databázových systémů.

Z výše uvedené analýzy je zřejmé, že při nasazení aplikace s SQLite databází na Docker Swarm se vyskytne spousta problému a také omezení. Většinu těchto potíží řeší QuestDB. Tato databáze má sice při menším počtu kontejnerů o něco větší nároky na provoz, naproti tomu ovšem při větším počtu kontejnerů jsou již nároky menší. Zároveň se ale ukázalo, že QuestDB nemá až tak velké nároky jak bylo předpokládáno. Navíc má více možností pro optimalizaci a zálohování dat.

Pokud aplikace bude sloužit pro sledování pouze několika kontejnerů (5-10), je výhodnější použít SQLite. Při větším počtu kontejnerů se již vyplácí využit databázi QuestDB. Jelikož je počítáno i s možností sledování kontejnerů běžících ve Swarm módu a sledování i více kontejnerů, je za cenu o něco větších nároků na hardware, při realizacích s menším počtem kontejnerů, použita databáze QuestDB.

4.3 Volba platformy pro kontrolu metrik

Platforma pro kontrolu metrik bude sloužit nejen pro kontrolu metrik získaných z databáze, ale zároveň se bude využívat i jako přístupový bod pro konfiguraci a sledování celého systému. Se sledováním také souvisí odpovídání na dotazy front-end aplikace. Je tedy žádoucí, aby zvolená technologie umožňovala snadnou práci s daty a podporovala dotazování do zvolené databáze. Nutnou podmínkou je podpora více vláknového běhu a knihoven pro REST API. Vysokoúrovňový programovací jazyk s patřičným framework nástrojem podporujícím vývoj webových aplikací je tedy vhodnou volbou pro tyto potřeby.

4.3.1 Express Framework

Node (Node.js) je open-source běhové prostředí pro vytváření serverových nástrojů a aplikací založených na technologii JavaScript. Běhové prostředí je určeno pro použití mimo kontext prohlížeče, běží tedy přímo na počítači nebo serveru. [48]

Zatímco Express je framework pro vývoj Node.js webových a mobilních aplikací. Jedná se o minimalistický a flexibilní framework obsahující vše potřebné pro vývoj single-page, multi-page a hybridních webových aplikací. Express obsahuje knihovny pro práci s cookie, autentifikací, URL, routing a mnoho dalších knihoven, které ulehčující a zrychlují vývoj. Běžnou webovou aplikaci lze díky tomuto nástroji jednoduše naprogramovat pomocí několika řádek kódu. [46, 48]

4.3.2 Python Django

Django je open-source webový framework pro platformu Python implementující návrhový vzor MTV (Model template view). Stejně jako ostatní zmíněné frameworky se snaží zjednodušit vývoj webových aplikací a obsahuje vše potřebné jako například knihovny pro autentifikaci, HTTP a ORM. Velkou prioritu má v tomto frameworku bezpečnost, což dokazuje také implementace jednoho z nejlepších zabudovaných bezpečnostních systémů. Django je preferován pro vývoj škálovatelných aplikací v omezeném čase v případě, kdy není nutné řešit vysoký výkon. V opačném případě, kdy je výkon vyžadován, je výhodnější použít již zmíněný Express. Jednou z výhod tohoto frameworku je tzv. batteries included (vše potřebné pro vývoj je již obsaženo a není tak potřeba dalších softwarů třetích stran) [46, 48]

4.3.3 Spring

Spring je framework používající platformu Java pro vývoj aplikací. Funkce tohoto frameworku jsou využitelné nejen pro samostatné aplikace, ale i pro

vývoj komplexních webových služeb. Spring implementuje mimo jiné dva zásadní koncepty, a to IoC (Inversion of Control) a DI (Dependency injection) pro odstranění závislostí v kódu, které značně zlepšují čitelnost a přehlednost kódu. IoC je princip přesunutí zodpovědnosti, nebo její části do frameworku, v tomto případě Springu. DI je návrhový vzor, díky němuž jsou objekty volně propojené. Tento vzor si můžeme představit tak, že si objekt vyžádá další objekty, na kterých je závislý. Jádrem celého spring frameworku je IoC kontejner, kde jsou Java objekty vytvořeny, konfigurovány a udržovaný v průběhu jejich životního cyklu. Spring se dělí do několika modulů: Core, Data Access, Web, Integration, Testing. Hlavní výhodou Springu je pružnost, široká podpora rozšiřujících knihoven a hlavně zjednodušuje psaní webových aplikací na platformě Java. [45, 46, 47]

Spring boot je rozšíření samotného frameworku Spring postavené na MVC architektuře. Obsahuje nejen samotné funkce a výhody Springu, ale také přináší další funkcionality. Mimo jiné obsahuje server Tomcat, který nám při spuštění projektu vytvoří HTTP webový server. Není tak nutné nastavovat Spring server manuálně, ale je nastaven Spring bootem automaticky. Nastavení v samotném frameworku Spring je velice zdlouhavé a i spuštění jednoduché webové aplikace zabere spoustu času. Spring boot tento problém řeší pomocí zavedení anotací a autokonfigurace. [45, 46, 47]

4.3.4 Shrnutí

Jelikož back-end nebude provádět příliš složité výpočetní operace, není od něj vyžadována příliš vysoká výkonnostní optimalizace a je zde široká škála využitelných frameworků. Každý ze jmenovaných frameworků je možné využít pro potřeby této práce a je zcela postačující. Python se vyznačuje svou jednoduchostí, Note.js zase rychlostí a v případě Javy se jedná o komplexnost a široké spektrum knihoven. Při výběru back-end softwaru je častým faktorem případ užití a preference jazyka, který je programátorovi bližší. Na tomto projektu je zvolena technologie Java Spring, primárně z důvodu předchozích zkušenosti s tímto frameworkem. V případě neznalosti ani jedné technologie by se jako lepší volba mohla nabízet varianta python Django, a to pro jeho krátkou syntaxi a přehlednost.

4.4 Front-end aplikace

Na aplikaci front-end nejsou kladeny příliš vysoké požadavky na výkon. Nejedná se o část, která bude nějak zvláště vytížená, jelikož se nepředpokládá velký nápor v podobě stovek uživatelů s potřebou sledovat metriky. Je ale zřejmé, že čím nižší zatížení bude mít server front-end aplikace, tím více zdrojů bude možno rozdělit mezi ostatní běžící aplikace. Je tedy nezbytné hledat jednoduchý a rychlý framework pro vývoj webových single-page aplikací, kde nízké nároky na server jsou jeho výhodou.

4.4.1 Angular

Angular je open-source MVVM framework pro vývoj single-page client aplikací. Byl vydán společností Google v roce 2016. Angular používá tři základní jazyky pro konstrukci aplikací, a to HTML, TypeScrip a CSS. HTML se zde využívá pro návrh, jakým způsobem prohlížeč zobrazuje prvky webové stránky. V TypeScript souborech je převážná část logiky aplikace a jazyk CSS je zde použit pro stylování HTML komponent. [49]

Architektura Angular je založena na komponentách organizovaných do bloků NgModules, které shromažďují související kód do funkčního celku. Celá aplikace je pak následně definovaná ze sad zmíněných NgModules a alespoň jedním hlavním root modulem. Komponenty využívají servisy, jež poskytují specifické funkce nesouvisející s pohledem a jsou do nich vloženy (depencency injection) jako závislosti. Komponenty i servisy se dají definovat jako třídy s dekorátorem, které poskytují metadata frameworku Angular. [49]

Jedná se o velice dobře dokumentovaný framework, a to i přes aktualizace a nové verze. Nejnovější verzí je v tuto chvíli Angular 11 vydaný v roce 2020. Nevýhodou Angularu je složitější výuková křivka pro začátečníky navzdory skvělé dokumentaci.

4.4.2 Blazor

Blazor je poměrně nový open-soruce webový framework vydaný v roce 2018 společností Microsoft. Jedná se o rozšíření platformy Mono, jenž je verzí .NET Frameworku, o nástroje a knihovny pro vytváření webových aplikací. Architektura Blazoru je založená na znovupoužitelných komponentách implementovaných pomocí jazyka HTML,C# a CSS. Microsoft oznámil několik různých edicí této aplikace, z nichž dvě jsou již vydané a jedná se o Blazor Server a Blazor WebAssembly. [50, 51]

V případě edice Blazor server vyžaduje aplikace k běhu server, na kterém se zpracovává logika. Události vyžadované klientem se odesílají serveru pomocí massaging frameworku a jejich odpověď v podobě požadované změny je odeslána nazpět. Klientský prohlížeč tak načítá jen malou stránku, a server tak přebírá velkou část zatížení. [50, 51]

Edice Blazor WebAssembly je oproti předchozí zmíněné edici odlišná v nárocích na server. V tomto případě je celá single-page aplikace stažena před používáním do klientského prohlížeče. Zpracování požadavků klienta se tak děje zcela na straně klienta a nezatěžuje server. Takovýto běh aplikací je možný díky jazyku WebAssembly běžícího v dnešních moderních prohlížečích namísto JavaScriptu. WebAssembly umožňuje běh kódu napsaného v různých jazycích jako C,C++, java a dalších včetně jazyka C#. [50, 51]

4.4.3 Shrnutí

Jedna z edic Blazoru s názvem Blazor WebAssembly je velice zajímavá pro potřeby této práce, a to díky jejímu běhu přímo v prohlížeči. WebAssembly edice redukuje nároky na server a využívá výkonu klientského zařízení. V případě potřeby co nejnižšího zatížení serveru se tato vlastnost velice hodí. Na druhou stranu ale také musíme zmínit delší načítání při otevření stránky, způsobené během programu u klienta, a to díky stahování celého webu (knihovny pro .net a dll).

Angulár je další možnost pro tuto práci s výhodou stavu production-ready. Naproti tomu Blazor je zatím stále ve vývoji, a tudíž se stále mění. Neopomenutelným faktem je rovněž podpora širší škály prohlížečů, jelikož WebAssembly podporují jen některé prohlížeče. Jedná se ale o nekompatibilitu pouze u méně používaných prohlížečů. U většiny známých jako Chrome, Firefox a Edge již WebAssembly lze spustit.

V konečném důsledku byl pro tuto práci vybrán Blazor, a to díky prioritizování nízkých nároků na server, a také vývojovým prostředím v podobě jazyka C#.

Kapitola 5

Návrh a implementace

Obsahem této kapitoly je primárně návrh systému pro monitorování metrik na základě zjištěných poznatků z předchozích kapitol. Z počátku se kapitola věnuje způsobu, jakým byly metriky získány, společně s popisem struktur. Dále již přichází na řadu samotný návrh architektury, znázorněný pomocí Diagramu komponent, včetně vzájemné komunikace aplikací. Z velké části se kapitola věnuje rozboru jednotlivých aplikací a jejich implementaci. Nechybí též představení uživatelského prostředí front-end aplikace. Na závěr je rozebrán způsob kontejnerizace pomocí Docker nástroje a pokročilé možnosti konfigurace, které systém nabízí.

5.1 Získání metrik

Jak již bylo nastíněno v sekci viz 2.4.2, metriky jsou získávány pomocí pravidelného čtení pseudo-souborů v Cgroups, které poskytuje Linux kernel. Existují také metriky, které tímto popsaným způsobem nezískáme, jelikož se v souborovém systému Cgroups nevyskytují.

5.1.1 Cgroups metriky

V pseudo-souborové struktuře Cgroups se vyskytuje mnoho souborů, ale pro potřeby monitorování metrik jsou využity pouze následující zmíněné pseudo-soubory (Pro zjednodušení nebudou uvedeny úplné cesty, ale pouze zkrácené. Budeme uvažovat cestu k Cgroups /cgroups a dále zkratka ID značící Docker ID kontejneru).

- `/cpuacct/docker/ID/cpuacct.stat` — obsahuje akumulované metriky CPU od spuštění procesu viz 1. Hodnoty jsou měřeny v milisekundách (pro x86 jsou měřeny v násobcích 10 milisekund).

```
user 75531 # čas strávený v uživatelském prostoru
system 1616 # čas strávený v kernel prostoru
```

Výpis kódu 1: Příklad souboru cpuacct.stat

- `cgroup/cpuacct/docker/ID/cpuacct.usage` — obsahem je akumulovaná hodnota celkového času využití CPU. Tato hodnota je udávána v nanosekundách bez dodatečného textu 2.

```
5899567100 # nanosekund od startu programu
```

Výpis kódu 2: Příklad souboru cpuacct.usage

- `/cgroup/cpu/docker/ID/cpu.stat` — v případe nastavení limitů na využití CPU může být kontejneru omezen přístup k CPU. Tento pseudo-soubor sleduje toto překročení 3.

```
nr_periods 0 # počet uplynulých případů omezení
nr_throttled  0 # počet případů omezení
throttled_time 0 # celkový čas omezení v nanosekundách
```

Výpis kódu 3: Příklad souboru cpu.stat

- `/cgroup/memory/docker/ID/memory.stat` — tento pseudo-soubor obsahuje podrobné informace o využití paměti v bytech. Je rozdělen na dvě části. V první jsou data pouze bez podskupin. Ve druhé části jsou již do dat zahrnuty i podskupiny procesu (metriky označené předponou total_). Vzhledem k velikosti celkového souboru budou nastíněna pouze data, které se pro monitorování využívají 4.

```
total_cache 11492564992
total_rss 1930993664
total_swap 0
hierarchical_memory_limit 92233720 # maximální fyzická paměť
hierarchical_memsw_limit 92233720 # maximální RAM + swap
```

Výpis kódu 4: Příklad souboru memory.stat

- `/cgroup/memory/docker/ID/memory.usage_in_bytes` — obsahuje součet využití cache a rss paměti. Obsahem je stejně jako u cpuacct.usage číslo, které je vyjádřeno v bytech.
- `/cgroup/memory/docker/ID/memory.memsw.usage_in_bytes` — obsahuje využití celkové paměti sečtené s swap pamětí, tedy cache + rss + swap. Hodnota je vyjádřena v bytech a bez textu.
- `/cgroup/memory/docker/ID/memory.failcnt` — obsahuje počet, kolikrát bylo dosaženo nastaveného paměťového limitu.

- `/cgroup/memory/docker/ID/memory.limit_in_bytes` — obsahuje hodnotu udávající paměťový limit. V případě dosažení se začne používat swap. Hodnota je vyjádřena v bytech a bez textu.

- `/cgroup/blkio/docker/ID/blkio.throttle.io_service_bytes` — obsahuje akumulovaný počet I/O operací v bytech. První číslo udává unikátní ID zařízení. Následně obsahuje zmíněný počet I/O operací, která nás zajímá viz následující příklad 5

```
8:0 Read 12238848
8:0 Write 0
```

Výpis kódu 5: Příklad souboru blkio.throttle.io_service_bytes

5.1.2 Síťové metriky

Pro získání síťových metrik nelze využít Cgroups jako v případě ostatních metrik, jelikož tyto metriky neobsahuje. Pro jejich získání existuje pseudo-souborový systém zvaný proc. Tento systém obsahuje informace o běžících procesech, jejichž součástí jsou i metriky využití sítě daným procesem. Pro získání síťových metrik je potřeba zjistit nejprve PID (proces id).

PID lze získat pomocí Cgroups, příkazu `docker inspect` anebo prostřednictvím Docekr API. V tomto případě je použito Docker API v podobě dotazu `GET /$Docker_ID/top`, který nám vrátí XML odpověď obsahující PID daného kontejneru.

Následně síťové metriky nalezneme v pseudo-souboru `/proc/PID/net/dev`. Tento soubor obsahuje informace o všech rozhraních, které kontejner využívá v podobě akumulovaných hodnot v bytech viz 6. Vyobrazená ukázka výpisu souboru `dev` je pouhou jeho částí. Ve stejném formátu tento výpis poskytuje informace i o Transmit hodnotách.

```
Inter-|                   Receive                                |
face |bytes    packets errs drop fifo frame compressed multicast
eth0:  14863    135    0    0    0     0          0         0
lo:        0      0    0    0    0     0          0         0
```

Výpis kódu 6: Příklad souboru dev

5.1.3 Metrika zatížení procesoru

Poslední nutnou metrikou, kterou je potřeba získat, je celkové zatížení procesoru. Tento údaj je nutný pro zpracování metrik při výpočtu procentuálního zatížení hostitele kontejnerem. I v tomto případě je možné využít presudo-souborový systém proc, jelikož poskytuje potřebné informace o aktivitě jádra v pseudo-souboru `/proc/stat`. Příklad obsahu zmíněného souboru `stat` 7.

```
cpu  737577 16804 226449 4279092 5606 0 2620 0 0 0
cpu0 192479 4065 57335 4265844 5445 0 994 0 0 0
cpu1 174820 3935 55935 4387 70 0 818 0 0 0
cpu2 183284 4815 57704 4123 56 0 177 0 0 0
cpu3 186994 3988 55474 4737 34 0 630 0 0 0
intr 47771259 0 2259 0 0 0 0 0 0 52221 0 0 21700 0 0 0  ....
ctxt 92469762
btime 1647766966
processes 273797
procs_running 5
procs_blocked 0
softirq 34773943 1480910 3786630 36 197852 ...
```

Výpis kódu 7: Příklad souboru stat

Potřebné informace pro výpočet celkového zatížení CPU se nacházejí hned na prvním řádku. Tento řádek uvádí akumulované hodnoty stráveného času CPU nad vykonáváním různých operací. Postupně v řádku hodnoty symbolizují: provádění procesu normal v uživatelském módu, prováděni procesu niced v uživatelském módu, provádění procesů v kernel módu, nečinnost, čekání na i/o, přerušení, softirqs, stolen time (strávený čas na jiném os v případě virtualizace), čas běhu niced quest. Sečtením všech těchto časů dostáváme hodnotu, které odpovídá maximální možné době běhu procesu na CPU. Časové jednotky závisejí na architektuře hostitele a jsou uvedeny v USER_HZ nebo Jiffies (ve většině případu 1/100 sekund). Pro snadnější výpočet tohoto času lze využít C funkci `sysconf(_SC_CLK_TCK)`, které nám vrátí hodnotu symbolizující počet tiků hodin procesoru za sekundu.

5.2 Databázový model

Jelikož se nejedná o standardní relační databázi, je nezbytné při návrhu brát tento fakt v potaz. V QuestDB databázi nenajdeme vazby mezi tabulkami, primární klíče ani zajištění unikátnosti záznamu či auto-inkrementaci. Se všemi těmito obtížemi si je nezbytné poradit při návrhu schématu databáze.

QuestDB je určena, jak již bylo zmíněno, na ukládání dat s časovou značkou. Tato časová značka se chová jako unikátní atribut, dle kterého se dá následně hodnota vyhledat. Můžeme jí tedy vnímat jako primární klíč. Je označena datovým typem timestamp, který v sobě obsahuje informaci jak datumovou, tak i časovou s přesností na setiny sekundy. Časové značka slouží nejen k identifikaci, ale také k fyzickému rozdělení nasbíraných metrik do souborů. Velikost těchto souborů, potažmo úseků dat uložených v jednom souboru, se dá specifikovat při vytvoření tabulky pomocí notace PARTITION BY a následně z nabídky YEAR, MONTH, DAY, HOUR. Takovýto přístup následně redukuje IO operace s diskem a rovněž značně ulehčí databázi při vyhledávání či agregaci záznamů dle data. Pro potřeby této práce bylo rozdělení

naměřených metrik zvoleno po dnech.

Dále je zapotřebí uchovat informaci o tom, ke kterému kontejneru se daná zaznamenaná hodnota vztahuje (cizí klíč). Pro tento problém databáze poskytuje speciální datový typ s názvem symbol. Jedná se o řetězcovou hodnotu, která není uložena přímo ve tvaru řetězce, ale namísto něj se uloží číslo s tímto řetězcem spojené. Tento přístup je vhodný, jelikož symbol je jediný datový typ, nad kterém se dají vytvořit indexy. Což při častém vyhledávání metrik dle konkrétního kontejneru je žádoucí.

V poslední řadě je potřeba vyřešit tabulku s kontejnery a jejich informacemi a tedy také volbu unikátního identifikátoru pro každý kontejner. Jelikož databáze neobsahuje auto-inkrementaci, nelze využít s postupem času narůstající hodnotu. Do databáze může vkládat více aplikací pro sběr metrik své nalezené kontejnery a nelze zajistit, díky neumožnění databáze transakčního zpracování, unikátnosti těchto inkrementů. Tento nastíněny problém může pomoci vyřešit samotný Docker. Dle článku [52] a současně dle dokumentace Docker je pro identifikaci kontejneru možné využít *container name* (–name), které je unikátní (v detailu kontejneru je dále označeno pod atributem image). Tento údaj je vhodný díky jeho unikátnosti a dá se tady použít v databázi jako symbol a rovněž pomyslný primární a cizí klíč pro tabulku s kontejnery a metrikami.

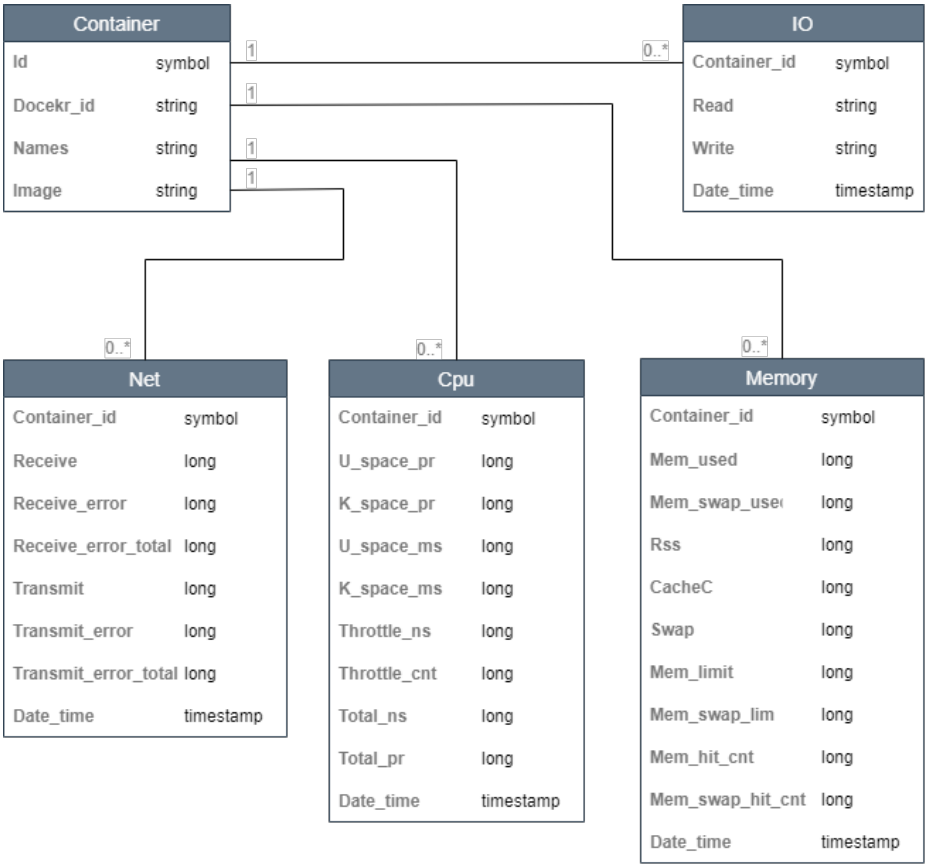
Pro každou metriku zvlášť není vhodné vytvářet samostatnou tabulku, jelikož čtení metrik probíhá po skupinách dle komponenty, na které metriky sledujeme. V případě velké tabulkové granularity se zvětší objem uložených dat o redundantní hodnoty a navíc se zpomalí vyhledávaní, jelikož je pravděpodobné zobrazovat (vyhledávat) metriky dle komponenty. Budeme tedy rozeznávat čtyři skupiny metrik: CPU, paměť, síť a IO. Každá tato skupina bude obsahovat symbol popisující kontejner, ke kterému se vztahují metriky ve skupině a časovou značku označenou datovým typem timestamp.

Návrh databáze beroucí v potaz předchozí úvahy je viditelný na diagramu: 5.1. Nastíněné vazby mezi tabulkami jsou pouze pro vizualizaci, v databázi nejsou žádné omezení nebo pravidla podmiňující tyto vazby.

5.3 Návrh monitorovacího systému

Pro budoucí rozšíření a přehlednost bude celý systém rozdělen do několika nezávislých aplikací/komponent, které mezi sebou budou komunikovat pomocí předem definovaného rozhraní. Takovéto striktní oddělení pomůže nejen při samotném provedení monitorování, ale také umožní komponenty v budoucnu vyměnit nebo upravit bez nutnosti modifikace kompletního systému.

Při návrhu monitorovacího systému je zapotřebí brát v potaz možnost budoucího rozšíření o monitorování Dockeru i ve Swarm módu. Docker ve Swarm módu je orchestrační nástroj, který umožňuje správu kontejnerů napříč více hostitelskými zařízeními viz 2.5. Jelikož metriky jsou zaznamenávány pomocí



Obrázek 5.1: Databázový diagram

Cgroups 2.4.2, dokáže aplikace pro čtení metrik sledovat pouze ty metriky kontejnerů, běžící na stejném hostiteli. Budoucím cílem je ale sledovat všechny kontejnery, nejen na jednom hostiteli. Proto tato aplikace pro čtení metrik musí běžet na každém hostiteli zvlášť a odesílat metriky dále do společného úložiště. Zmíněné předpoklady nám již udávají první části systému a to samostatnou aplikaci pro čtení metrik, dále nazývanou jako Collector aplikace.

Další částí je již zmíněné uložiště v podobě QuestDB databáze. Uložiště bude sloužit pro uložení všech zaznamenaných metrik, ale zároveň i obsahovat základní informace o kontejnerech viz diagram 5.1. Jelikož QuestDB se nezabývá příliš bezpečností, její rozhraní pro komunikaci jsou špatně anebo vůbec chráněná a proto není vhodné je vystavovat jinde, než na lokální síti. Díky předpokladu sledování a vizualizaci metrik vzdáleně s patřičným zabezpečením, je nutné tento problém s bezpečností vyřešit. Nabízí se možnost vytvoření další aplikace jako prostředníka mezi databází a webovou aplikací. Díky čemuž odstíníme databázi od okolního světa s možností vlastní volby zabezpečení. Tuto aplikaci budeme dále označovat jako Monitor aplikace.

Monitor aplikace bude sloužit nejen jako bezpečné rozhraní pro komunikaci s databází, ale rovněž bude obsahovat samotné monitorování kontejnerů na základě uživatelem definovaných pravidel uložených v XML struktuře. Součástí bude také notifikační modul s integrací komunikačního nástroje Slack.

Poslední pomyslnou částí celého systému je front-end aplikace sloužící pro vizualizaci a konfiguraci systému. Bude komunikovat s Monitor aplikací přes zabezpečené rozhraní díky nutnosti autentifikace uživatele. Dále tuto aplikaci budeme označovat jako Viewer aplikace.

Celý návrh systému, jak byl popsán výše, je možné vidět na diagramu 5.2. Mimo jiné obsahuje i popis protokolů a rozhraní, které jednotlivé aplikace používají pro komunikaci mezi sebou.

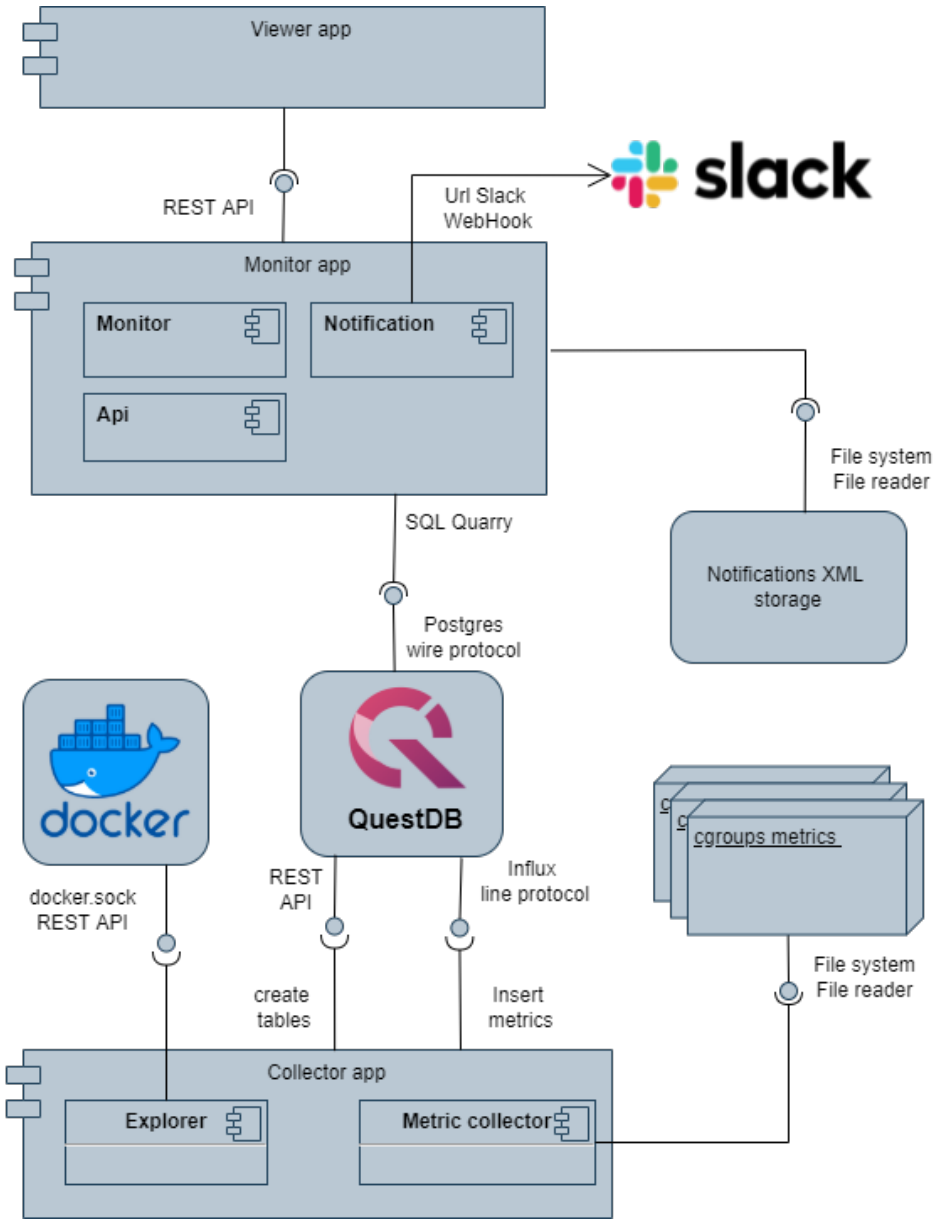
Z diagramu 5.2 je možné si povšimnout izolace Collector aplikace a potažmo i Dockeru od zbytku systému. Tato realizace má několik důvodů. Prvním, ne příliš zásadním, důvodem je potencionální běh více aplikací Collector. Aplikacím Collector je tak obecně obtížnější nastavit pevnou adresu či port, na kterém budou poslouchat na rozdíl od ostatních aplikací, jenž běží pouze jako jedna instance. Druhým, závažnějším, důvodem je bezpečnost. Aplikace Collector mají nejen přímý přístup k Docker daemonu a mohou tak ovládat celý Docker, ale také mají přístup do samotného souborového systému Cgroups nacházejícího se na hostiteli. Proto je vhodné tuto aplikaci zcela odstínit od zbytku systému pro vyhnutí se potencionálnímu riziku. V případě prolomení obrany tak útočník může získat monitorovaná data kontejnerů a potencionálně rozbít notifikace, ale k ovlivnění samotných monitorovaných kontejnerů dojít nemůže.

5.4 Collector aplikace

Jedná se o aplikaci, které bude nasazena na všech hostitelích s běžícím nástrojem Docker, kde je záměrem sledovat metriky kontejnerů. Collector aplikace komunikuje s Dockerem, databází a psoudo-souborovým systémem na hostiteli viz popis rozhraní 5.4.1.

5.4.1 Rozhraní

- **Docker REST** — Pro získání všech běžících kontejnerů na hostiteli je nezbytný přístup k `docker.sock`. Jedná se o rozhraní, prostřednictvím kterého lze řídit celý Docker. Komunikace probíhá na základě HTTP protokolu. Pro získání listu kontejnerů lze využít http příkaz `GET /containers/json`, který vrátí v http response body list kontejnerů formátovaných v json formátu. Pro výpis pouze běžících kontejnerů je příkaz rozšířen o parametr `status` roven hodnotě `running`. V aplikaci Collector je pro HTTP komunikaci využita knihovna cURL. Dále pro parsování odpovědi byla využita knihovna rapidjson.



Obrázek 5.2: Diagram komponent

- **QuestDB REST** — Jelikož databáze QuestDB v sobě nemá zabudovanou funkci automatického vytvoření databáze z předdefinovaného schématu, je zapotřebí tabulky vytvořit manuálně. Tento krok je nutný pouze při prvotním startu systému. Pro vytvoření tabulek v databázi je použito rozhraní REST API, na které lze zaslat SQL příkazy. QuestDB otevírá standardně port 9000, na kterém se ono rozhraní nachází. Pod-

poruje nejen exekuci SQL příkazů, ale rovněž import CSV dat nebo jejich export. Pro exekuci SQL příkazů se používá URL `/exec` s URL parametrem `query` jehož hodnota je rovna našemu požadovanému SQL příkazu. Tímto způsobem můžeme databázi zaslat požadavky na vytvoření tabulek. Komunikace je také zprostředkována pomocí knihovny cURL.

- **QuestDB Influx line protocol** — Rozhraní je přímo určeno pro vkládání dat do databáze, ačkoli se dá využít i REST, je tento způsob vkládání vhodnější. Oproti REST rozhraní je navrženo pro práci s velkým množstvím dat od více zdrojů najednou. Influx line protocol je text-based formát, který je do databáze QuestDB odeslán prostřednictvím TCP nebo UPD. Struktura protokolu 14.

```
table_name,symbolset columnset timestamp\n
```

Výpis kódu 8: Struktura Influx line protokolu

Prvním atribut je název tabulky, do které se data vkládají. Následujícím znakem je čárka a výčet symbolů ve tvaru název=hodnota oddělených čárkou. Dalším znakem je mezera následovaná výčtem ne-symbolových atributů také oddělených čárkou. Protokol je zakončen mezerou následovanou časovou značkou a symbolem určujícím novou řádku `\n` viz ukázka.

5.4.2 Moduly

Collector aplikace je rozdělena do několika modulů za účelem snadné rozšiřitelnosti a přehlednosti. Hlavní funkcionalitu plní dva moduly Container Explorer a Metric Collector. Dále je zde několik menších modulů pro konfiguraci, parsování nebo komunikaci.

5.4.2.1 Container Explorer modul

Modul Container Explorer poskytuje funkce pro objevení nově spuštěných kontejnerů a odstraňování již ukončených. Mimo jiné také inicializuje a připraví kontejner pro následné čtení metrik. Tento modul pokrývá požadavek F1.1.

Nejprve se načte seznam běžících kontejnerů pomocí Docker API. Seznam kontejnerů je v json formátu a proto je nejprve potřeba provést samotné parsování do objektů. Následně jsou odstraněny kontejnery, které jsou vyloučeny z monitorování na základě konfiguračního souboru ini. Kontejnery je dále nutné inicializovat před samotným čtením metrik. Objevování kontejnerů probíhá opakovaně a není nutné inicializaci provádět opakovaně, proto jsou inicializovány pouze nově objevené kontejnery.

Inicializace kontejneru probíhá v několika krocích. Nejprve se obecné informace o kontejneru vloží do databáze. Dále se zjišťují dodatečné informace o kontejneru, jako je například PID. V poslední řadě se vygenerují veškeré

cesty k metrikám na základě zjištěných informací o kontejneru. Po této operaci je kontejner připraven pro čtení metrik.

5.4.2.2 Metric Collector modul

Jak již samotný název napovídá, tento modul zajišťuje funkcionalitu pro čtení a také ukládání metrik kontejnerů v rámci požadavků F1.2., F2.2. a F2.3..

V první fázi čtení jsou přečteny globální metriky, nesouvisející s konkrétními kontejnery. Následně v další fázi přichází na řadu samotné čtení metrik kontejnerů. Samotné čtení probíhá nejprve průchodem uvedeného seznamu kontejnerů s cílem přečíst cesty k metrikám. Na základě těchto cest se již přečtou všechny soubory s metrikami. Výsledkem čtení je řetězcová hodnota, která prochází procesem pársování. Po úspěšném přečtení a parsování všech metrik se dále metriky zpracují. Zpracování metrik probíhá nejprve odečtením akumulovaných metrik od předchozího čtení a dále vypočtením procentuálních hodnot a konverzí viz 5.1. Před uložením dat do databáze prostřednictvím Inflix line protokolu se metriky opatří časovou značkou.

5.5 Monitor aplikace

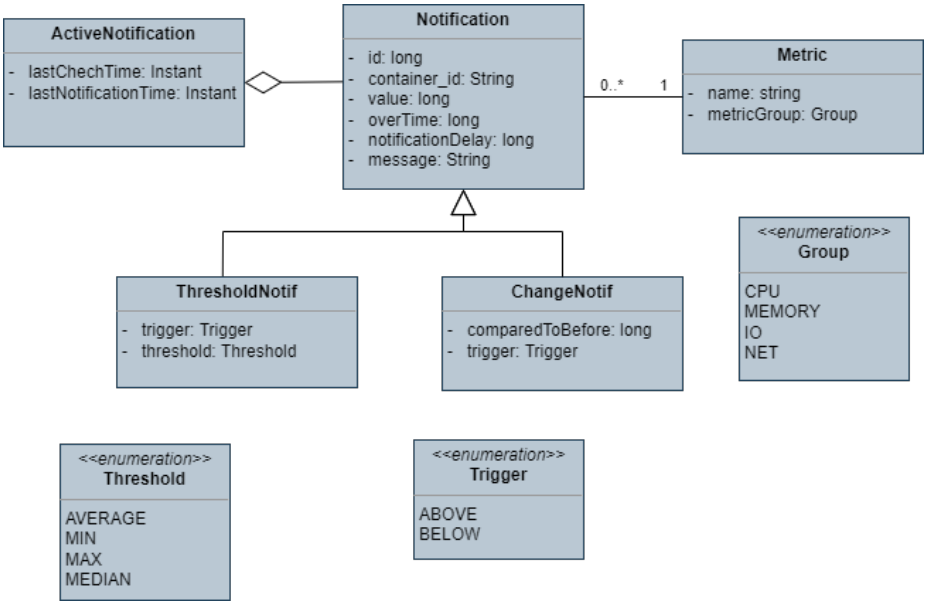
Aplikace Monitor slouží v celém systému nejen jako monitorovací prostředek, ale též jako rozhraní pro přistup k datům a konfiguraci. Při návrhu této aplikace byl použit návrhový vzor MVC (Model-View-Controlle). Jedná se o často používaný vzor u webových aplikací.

MVC rozděluje logiku programu a uživatelské rozhraní do tří vzájemně propojených komponent Model, View a Controller. Model zpracovává byznys logiku celé aplikace. Obsahuje objekty a data specifické pro danou doménu. View obsahuje rozhraní pro prezentaci dat získaných od Controlleru. Poslední komponenta Controller je mezičlánek přijímající požadavky od klienta pro které zajistí potřebnou akci. Zjištěné informace následně pošle do komponenty View, kde se výsledek zobrazí uživateli. [55]

5.5.1 Diagram tříd

Struktura tříd s metrikami je velice jednoduchá a přímočará, proto není zapotřebí jí znázornit pomocí Diagramu tříd. Její struktura je velice obdobná jako databázový diagram viz 5.1.

Struktura metrik není jediná struktura tříd, jelikož pracujeme také s notifikacemi. Diagram znázorňující strukturu pro notifikace, je znázorněn pomocí jazyka UML na obrázku 5.3. Z diagramu jsou pozorovatelné třídy společně s jejich nezbytnými atributy a stavy, ve kterých se mohou vyskytovat.



Obrázek 5.3: Diagram tříd notifikace

5.5.2 Moduly

Obdobně jako v případě Collector aplikace je i aplikace Monitor rozdělena do několika modulů, které zajišťuji potřebné funkcionality.

5.5.2.1 Monitor modul

Monitor modul je nejnáročnější modul z hlediska výkonu celé aplikace Monitor. Zajišťuje funkcionalitu pro pravidelné kontrolování metrik a pokrývá tak požadavek F3..

Nejprve modul projde veškeré notifikace s cílem najít takové, které je potřeba vyhodnotit. Jedná se o notifikace, u kterých se jejich čas pro další periodickou kontrolu naplnil. Následně se projdou všechny takto zjištěné notifikace a načtou se pouze potřebná data z databáze pro všechny notifikace najednou. Zajistíme tak, že data nenačítáme pro každou notifikaci zvlášť, jelikož by mohlo dojít ke zbytečnému dotazování na data, která již byla načtena pro některou z předchozích notifikací.

Následně se znovu projdou veškeré označené notifikace pro kontrolu. Na základě načtených metrik v rámci požadovaných intervalů a uživatelem zvolených parametrů se rozhodne, zda je nutné odeslat upozornění či nikoli.

Na závěr se aktualizuje čas pro další notifikaci. V případě, kdy bylo odesláno upozornění, se aktualizuje rovněž čas další kontroly notifikace na zadanou hodnotu.

5.5.2.2 API modul

Jako zprostředkovatele a prostředníka mezi webovou aplikací a databází je použit v aplikaci Monitor právě API modul. Poskytuje funkcionality pro konfiguraci systému. Implementuje také rozhraní a konvertory, které umožňují získat data z databáze ve zpracované podobě. Pokrývá tak hned několik požadavků F4., F5., F6.2., F7. a F8..

Pro optimalizaci komunikace po síti umožňuje tento modul redukovat velikost dat. Tato redukce probíhá na základě zvolené časové přesnosti, v jaké chceme data obdržet. V případě, kdy nepotřebujeme sledovat všechny naměřené hodnoty, můžeme pracovat s průměrnou hodnotou v rámci zvoleného časového úseku. Výsledkem takového dotazu tak mohou být například agregované hodnoty reprezentujíc chování metriky v průběhu hodiny s hodinovým rozestupem v případě volby hodinového intervalu.

5.5.2.3 Notifikační modul

Modul notifikací implementuje funkci pro rozeslání informace o naplnění některého pravidla pro notifikaci. Plní tak požadavek F7.5.. Modul implementuje návrhový vzor observer. Všichni sledující, kteří chtějí dostat upozornění, při naplnění pravidla, se registrují jako sledovatelné a následně dostávají upozornění.

Jediným implementovaným observerem pro tuto chvíli je Slack. Tento observer díky webhooku dokáže informovat uživatele o naplnění pravidla. Součástí observeru je také formátování oznámení.

5.5.2.4 Authorizační modul

Při použití OAuth viz 5.5.4 je nutné využít Autorizační server vlastní nebo třetích stran. Tento modul přináší implementaci a funkcionality právě Autorizačního serveru. Jeho hlavní činností je zkontrolovat uživatelské údaje s vnitřní databází (v tomto případě pouze jeden uživatel) a přidělit klientovi acces token pro další komunikaci k chráněnému obsahu.

5.5.3 Rozhraní

- **Slack** — Pro odesílaní oznámení (notifikací) do Slack aplikace je použita knihovna jslack. Tato knihovna využívá incoming Webhook pro odeslání zprávy z externího zdroje do Slack aplikace prostřednictvím běžného HTTP požadavku.

- **QuestDB Postgres wire protocol** — Databáze QuestDB integruje Postgres wire protocol a lze tak s databází komunikovat díky existujícím knihovnám. S využitím java.sql.DriverManager lze odeslat databázi query s odpovědí ve tvaru ResultSet. Pro QuestDB neexistuje zatím

žádná knihovna s ORM, která by usnadnila manipulaci s daty (mapování na objekty) a je tak nutné řešit zpracování ResultSet do použitelných objektů. Persistence dat bude zajištěna pomocí vzoru Row Data Gateway. Tedy jeden řádek tabulky v databázi odpovídá jedné instanci. Pro vyhledávání a ukládání je vyhrazena speciální třída.

5.5.4 Bezpečnost

I v případě pouhého monitorování je vhodné klást důraz na zabezpečení a z tohoto důvodu je bezpečnost řešena i v rámci navrhovaného řešení. Pro přístup k monitorovacímu systému z vnějšku se využívá REST rozhraní aplikace Monitor. Zabezpečení se tedy bude týkat právě tohoto rozhraní.

Jedním z požadavků, viz požadavek N4., je povolení přístup do aplikace pouze za předpokladu zadání správného uživatelského jména a hesla, je tedy potřeba uživatele autentifikovat. Je také zároveň uživatelsky přívětivé automatické přihlášení uživatele, pokud byl uživatel již autentifikován, například v případě předchozího zavření nebo znovu načtení stránky.

5.5.4.1 API Autentifikace

Autentifikace je proces, při němž je cílem ověřit zda identita uživatele je stejná jako za kterou se vydává. Pro ověření je spousta možnosti v podobě například klíče, biometrického otisku, ale nejčastější ověření je pomocí uživatelského jména a hesla. Pro autentifikace/přihlášení prostřednictvím webové aplikace je možné využít z několika možných protokolů. Jedněmi z nejběžnějších protokolů jsou HTTP Basic Auth, HTTP Digest Access nebo OAuth.

- **Basic Authentification** — Jedná se o nejjednodušší autentifikaci v podobě odeslání zašifrovaného jména a hesla. Jméno a heslo je ve formátu Jméno:Heslo a následně zašifrováno pomocí Base64. Tento řetězec je přidán k požadavku v rámci hlaviček HTTP ve tvaru: „Authorization= Basic YWRtaW46aHVudGVyMg==“. Takovýto požadavek je následně zaslán serveru, kde je hlavička Authorization dekódovaná a rozhodnuto o udělení přístupu. Nejedná se o příliš bezpečný způsob, ale lze ho zlepšit pomocí použití https.

- **Digest Access** — Autentifikace probíhá nejprve přidělením serveru klientovi číslo (nonce), které klient zkombinuje se jménem, heslem a URI. Tento řetězec se zašifruje pomocí šifry MD5 a přidá se jako hlavička do požadavku. Na straně serveru se po příchodu požadavku aplikuje stejný proces. Výsledná šifra se následně porovná s autentifikační hlavičkou. Pokud jsou shodné, je přistup povolen.

- **OAuth** — Protokol aplikuje zcela nový přístup k autentifikaci a to tak, že jí nechává na třetí straně, které autentifikaci provede. Předpokladem

je, že třetí strana je důvěryhodný zdroj, kterému obě strany důvěřují. V případě úspěšné autentifikace u třetí strany je server obeznámen o výsledku a dále se pro autentifikaci mezi serverem a klientem využívá autentifikace pomocí token.

Zmíněné protokoly jsou dále podrobněji vysvětleny v knize [53].

Pro zabezpečení monitorovacího systému byl použit protokol OAuth resp. OAuth 2.0. Jedná se o dnes již široce používaný protokol pro zabezpečení a považuje se za jeden z nejbezpečnějších. Vzhledem k jeho použití bude dále tento protokol více rozveden. Informace o OAuth byly čerpány z knihy [54], kde je tento protokol detailněji vysvětlen.

OAuth definuje čtyři role:

- **Resource Owner** — Vlastník zdroje je systém nebo uživatel, který vlastní zdroje a může k ním udělit přístup.
- **Client** — Klient je systém, který vyžaduje přistup ke zdrojům. Pro přístup musí mít Acces token.
- **Resource Server** — Server se zdrojovými daty je server, který ověří identitu uživatele na základě access tokenu a odešle chráněné informace.
- **Authorization Server** — Autorizační server je třetí strana, která dokáže uživatele autentifikovat.

Pro provedení operace autentifikace v případě OAuth je nutné, aby klient byl zaregistrován u autorizačního serveru. Získá tak client_secret, který bude znát pouze klient s autorizačním serverem (zamezí se tak podvržení klienta). Přihlašování probíhá nejprve požadavkem klienta na autorizaci u autorizačního serveru. Součástí tohoto požadavku je client_id, client_secret, scope a redirection URI, kam se má odeslat access token. Po přesměrování na autorizační server, požádá server vlastníka o identifikaci pomocí přihlašovacích údajů. Po úspěšném přihlášení je zaslán access token klientu. Tento token je následně použít pro identifikaci vlastníka při komunikaci se serverem obsahujícím chráněné zdroje. Server obsahující zdroje zkontroluje platnost access tokenu a odešle chráněný obsah klientovi.

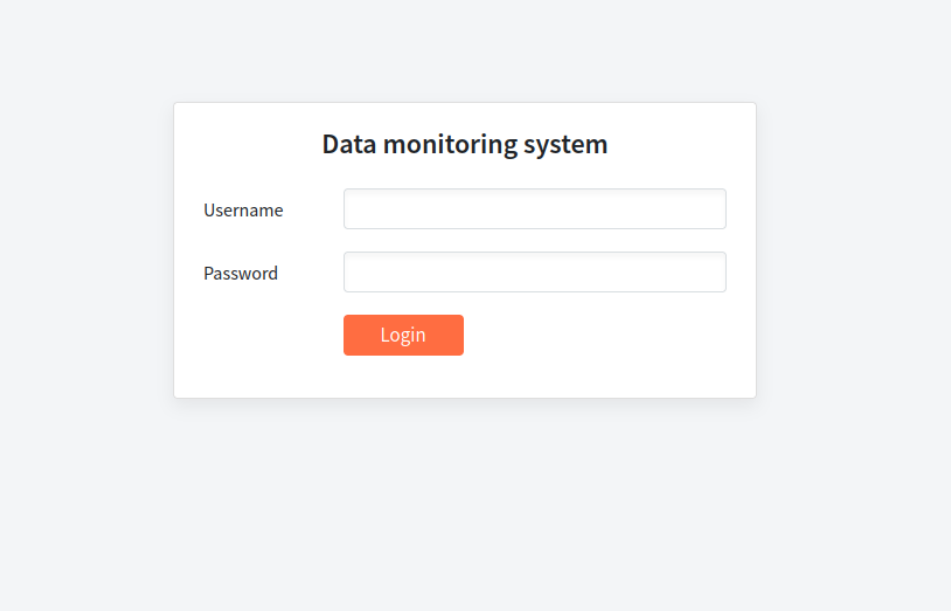
Jak již bylo zmíněno, aplikace Monitor má zabezpečené API pomocí OAuth protokolu. Za zmínku stojí způsob jeho nasazení. Většinou se předpokládá, že autorizační server je součástí třetí strany, je tedy nezávislý na systému. V této práci je ale implementován vlastní autorizační server. Díky tomuto řešení je možné zachovat systém zcela odstíněn od přístupu z veřejné sítě. Zároveň je jednodušší prvotní přihlášení pomocí výchozích údajů. Tento krok ale nelimituje nijak budoucí rozšíření, jelikož je možné systém rozšířit o jiný autorizační server.

5.6 Viewer aplikace

Architektura front-end webové aplikace Viewer je založena na konceptu znovupoužitelných komponent (Component Based Archytecture). Komponenty zapouzdřují funkčnost a chování do opakovaně použitelných jednotek zvaných komponenty. Díky tomu je možné vyvíjet aplikace rychleji a také s větší spolehlivostí, díky znovupoužitelnosti již ověřených komponent. Je zapotřebí se vyvarovat nadměrně velkým komponentám s velkou povinností. Proto je nutné stále dodržovat princip separation of concerns. [56]

5.6.1 GUI

Pro kontrolu metrik a konfiguraci systému je pro uživatele připraveno grafické rozhraní. Rozhraní je přístupné prostřednictvím webové aplikace na adrese http://0.0.0.0:80. Pro vstup do aplikace je nejprve nutné přihlášení pomocí jména a hesla viz obrázek 5.4. V případě správných údajů je uživatel přesměrován na základní stránku s dashboardem.



Obrázek 5.4: Přihlášení

Webová Stránka je rozdělena na dvě části. Na levé straně se nachází menu s nabídkou různých možností zobrazení nebo konfigurace. V pravé části je prostor pro zobrazí zvoleného obsahu. Menu obsahuje několik záložek: Dashboard, Containers, Notifications, Configurations, User profile a Data Removal.

Na stránce s Dashboad jsou zobrazeny veškeré kontejnery jako dlaždice se základními live informacemi o využití paměti a procesoru. Tento dashboard lze

pozorovat na obrázku 5.5. Po kliknutí na dlaždici je uživatel dále přesměrován na detail příslušného kontejneru.

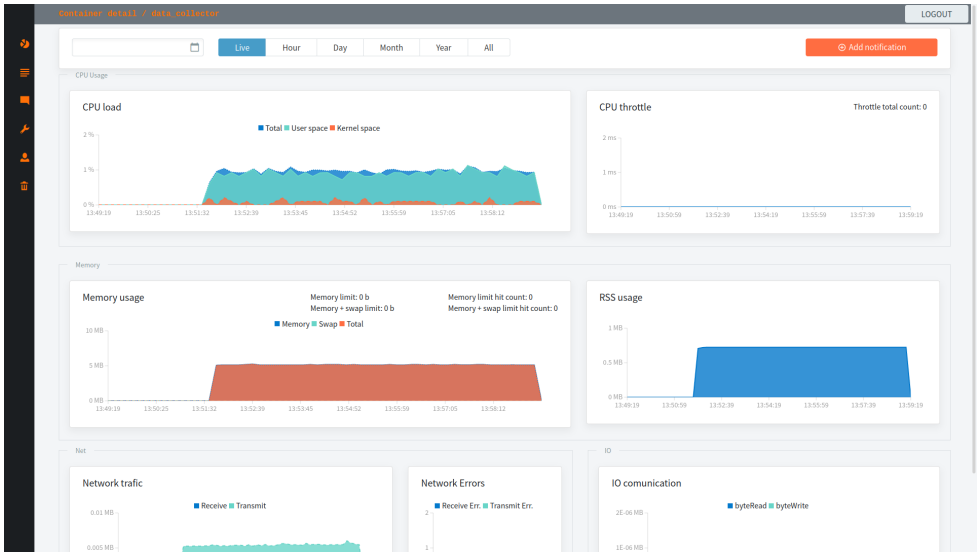
Obrázek 5.5: Dashboard

Stránka s detailem kontejneru zobrazuje podrobnější monitorované metriky v podobě grafů viz 5.6 V horní liště je možné změnit mód sledování z live na konkrétní časový okamžik a rozsah s jakým chceme metriky sledovat. Je možný výběr sledování v rozsahu hodiny, dne, měsíce, roku nebo od začátku měření. Dále se v této liště nachází tlačítko pro přidání nové notifikace 5.7, díky které můžeme nastavit upozornění na překročení zvolené metriky.

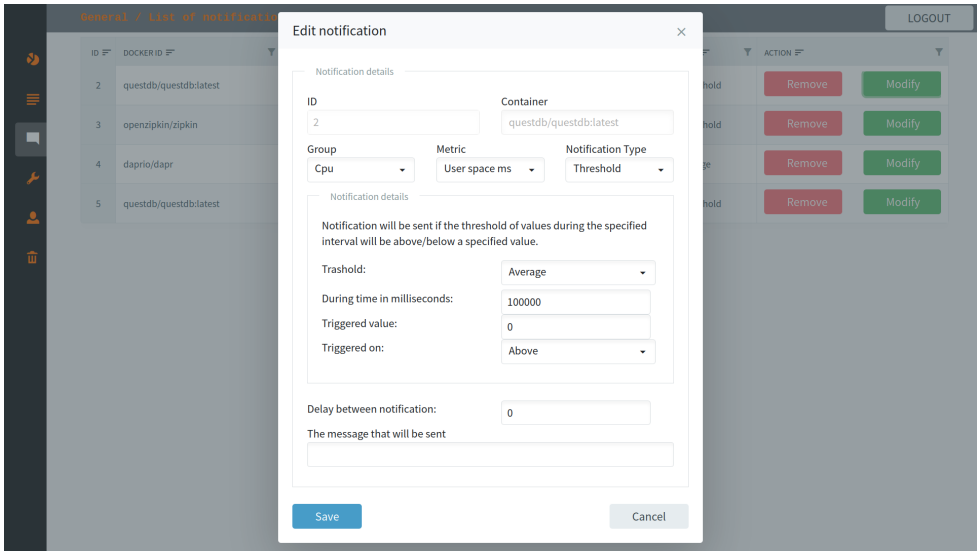
Po kliknutí na tlačítko přidání notifikace se zobrazí nové okno pro jejich vytváření. Nejprve je nutné zvolit skupinu metrik a až následně konkrétní metriku. Dále je na výběr z dvou módů sledování a to změnového nebo prahového viz 3.1.5. Mimo jiné je rovněž možné mimo časového nastavení zvolit zprávu, která bude při notifikaci odeslána společně s hodnotou, která přesáhla uvedenou hranici.

Záložka Container je velice obdobná jako zmíněný Dashboard. Také zobrazuje kontejnery, ale pouze jejich obecné informace v tabulce viz 5.8. Lze tak kontejner jednoduše vyhledávat v případě, kdy je sledováno mnoho kontejnerů, jelikož Dashboard muže být v takovém případě nepřehledný. Po kliknutí na kontejner v tabulce je uživatel přesměrován na detail příslušného kontejneru stejně jako v případě Dashboardu.

Pro přehled a manipulaci s notifikacemi slouží záložka Notification 5.9. Tato záložka zobrazuje seznam aktivních notifikací, které lze modifikovat nebo smazat po kliknutí na příslušné tlačítko. V případě modifikace se zobrazí obdobné okno jako při vytváření, pouze s předvyplněnými hodnotami, které je



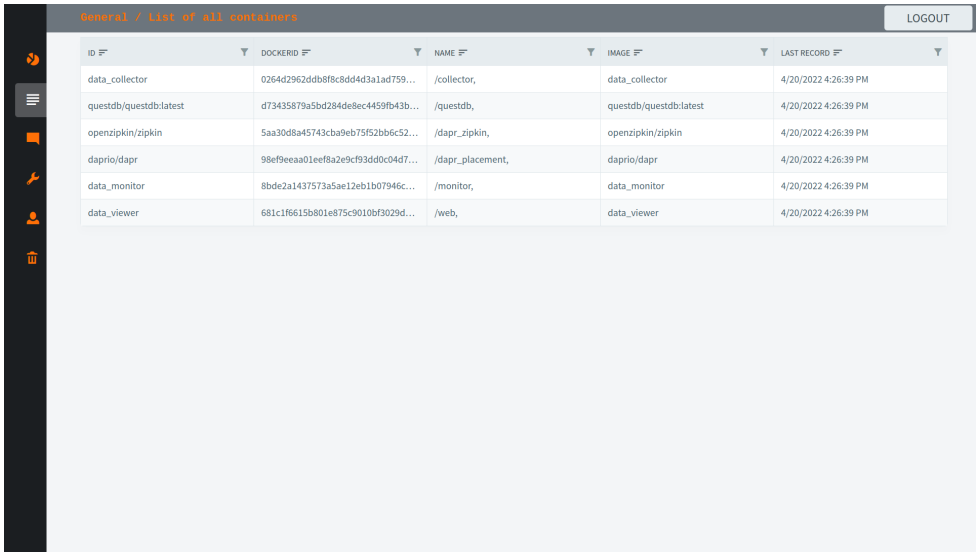
Obrázek 5.6: Detail kontejneru



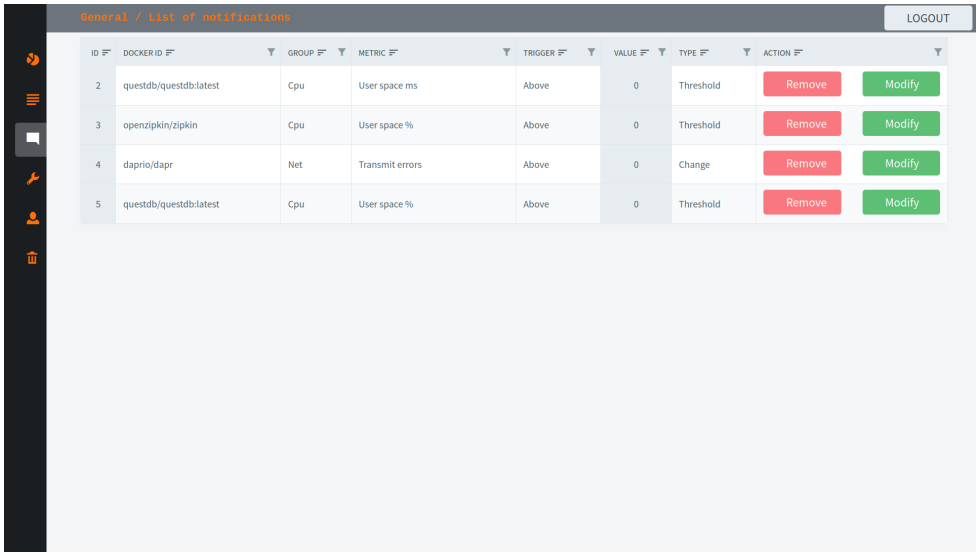
Obrázek 5.7: Editace / vytvoření nového pravidla

možné změnit.

Konfigurace Slack webhooku a také aktivaci notifikací je možné nastavit prostřednictvím další záložky v pořadí s názvem Configuration. 5.10. Dle zvoleného Slack webhooku se odesílají notifikace v podobě zpráv na správný Slack server 5.11.



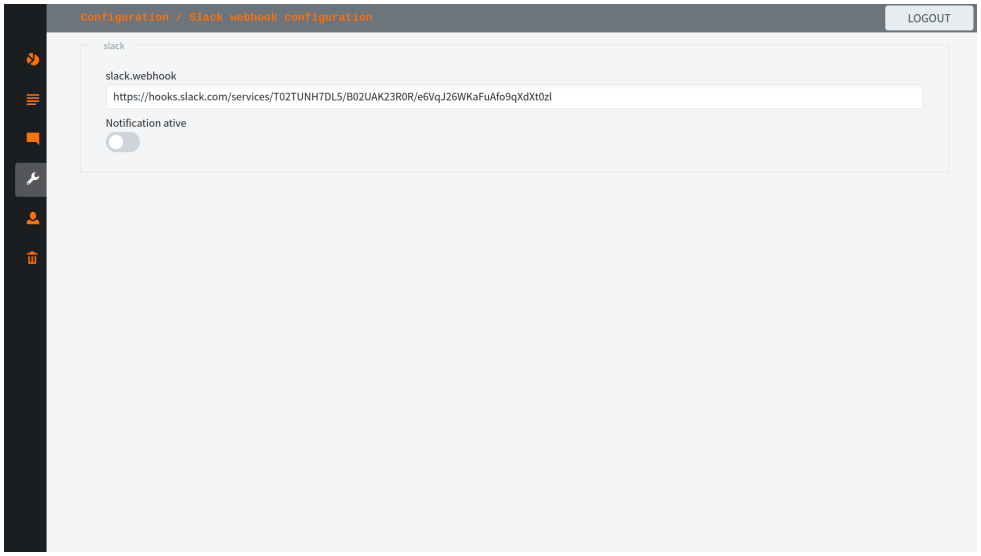
Obrázek 5.8: Seznam všech sledovaných kontejnerů



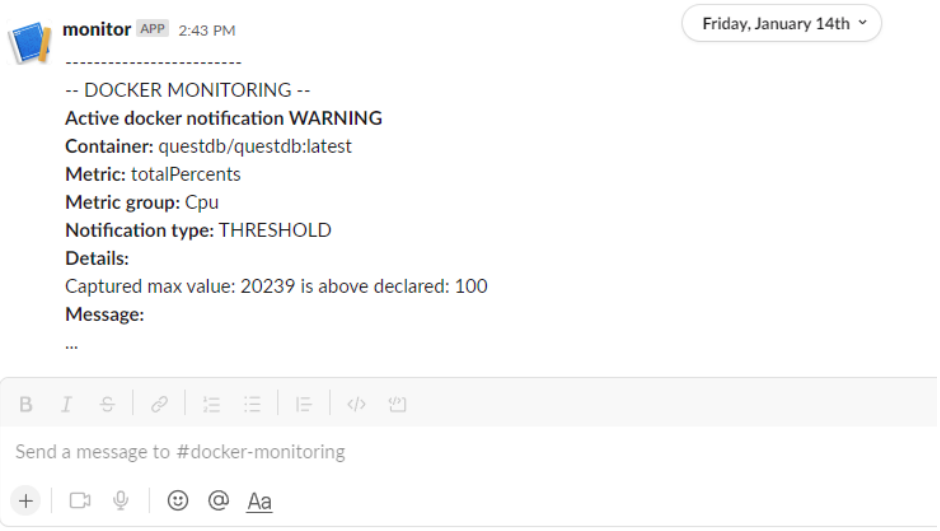
Obrázek 5.9: Seznam všech vytvořených pravidel pro notifikace

Další záložka v pořadí je User profil 5.12, prostřednictvím které je možné změnit jak uživatelské přihlašovací jméno, tak heslo. V případě prvotního spuštění jsou údaje defaultní a pro bezpečnost je vhodná jejich změna.

Poslední záložka, určená pro správu mazání kontejnerů a jejich metrik, je označena Data Removal 5.13. První tabulka zobrazuje veškeré kontejnery.

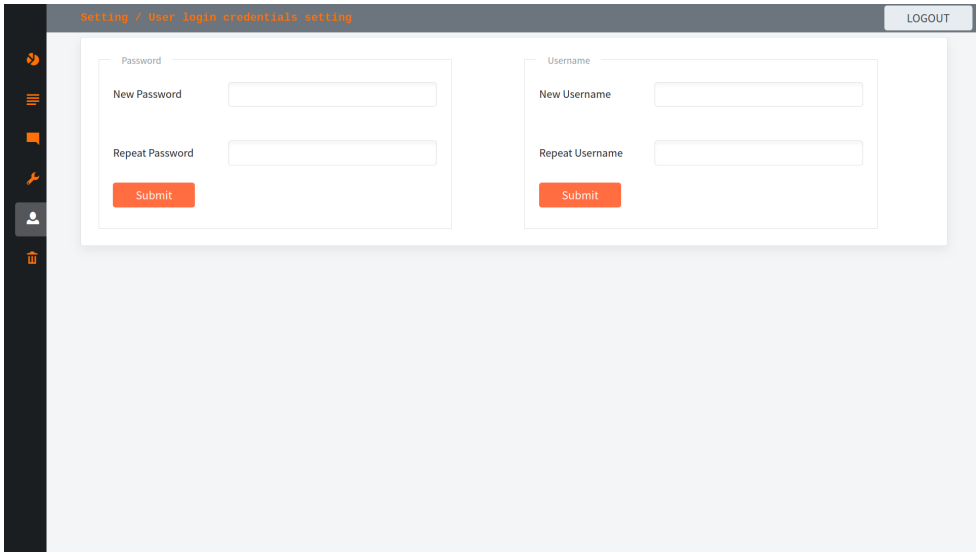


Obrázek 5.10: Konfigurace Slack



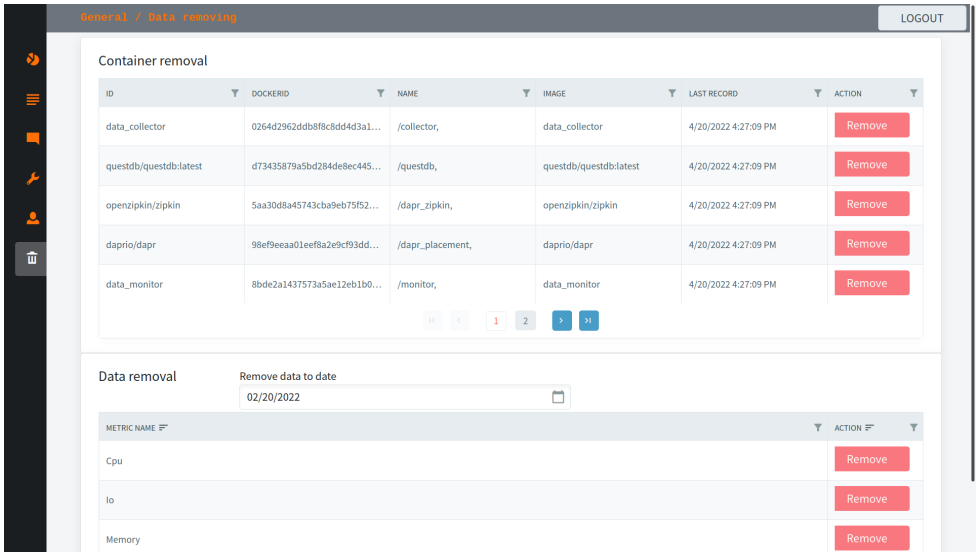
Obrázek 5.11: Slack notifikace

Z této nabídky je možné vybrat určité kontejnery ke smazání. Samotné metriky ke konkrétnímu smazanému kontejneru ale odstraněny nebudou. Pod tabulkou je dále možné mazat konkrétní metriky dle zadaného data. Datum označuje dobu, do které budou všechny metriky pro všechny kontejnery smazány. Nelze mazat pouze metriky pro jeden kontejner, díky omezení způsobené databází



Obrázek 5.12: Nastavení nového jména a hesla

QuestDB.



Obrázek 5.13: Odstranění metrik a kontejnerů

5.7 Kontejnerizace monitorovacího systému

Jedním z požadavků, viz požadavek N3., je nasazení systému do Docker kontejneru. Tento krok nejen ulehčí nasazení celého systému na hostitele, ale také celou konfiguraci.

Každá aplikace (Monitor, Collector, Databáze a Viewer) je zcela oddělená a bude běžet ve svém vlastním izolovaném kontejneru. Pro kontejnerizaci je nejprve nutné definovat image pomocí Dockerfile souboru 2.4.1.3. Jako základní image, z něhož se vychází, je vhodné použít co možná nejmenší a nejrychlejší distribuci linuxového prostředí. Jednou z takových to distribucí je Alpine s velikostí 5MB, jehož rozšířené verze byly použity jako základní image pro všechny aplikace monitorovacího systému.

Zdrojové soubory aplikací je nejprve nutné před spuštěním zkompilovat pro konkrétní distribuci, tedy v tomto případě Alpine. Pro samotnou kompilaci je nezbytné distribuci Linuxu rozšířit o prostředky (knihovny a nástroje) umožňující tuto kompilaci. Následně lze aplikace zkompilovat a spustit. Díky zmíněným prostředkům nezbytným pro kompilaci narůstá zbytečně velikost výsledného image. Proto je dobré využít tzv. multi-stage build, který tento problém řeší. Jedná se o proces, při kterém se nejprve vytvoří image, do které se přidají veškeré prostředky pro kompilaci, a následně se také kompilace provede. Po tomto kroku se spustitelná aplikace přesune do nového čistého image bez prostředků pro kompilaci a smaže se původní image. Výsledkem je velice malý image obsahující pouze nutná data pro běh aplikace. V případě, kdy aplikaci vyžaduje k běhu knihovny třetích stran, je nutné do výsledného image tyto knihovny přidat.

Celý monitorovací systém je následně spuštěn a konfigurován pomocí nástroje Docker Compose 2.4.1.4. V rámci tohoto nástroje jsou mimo jiné nakonfigurovány výstupní porty a svazky pro jednotlivé aplikace. Díky těmto portům je možné s kontejnerem/aplikací komunikovat.

Přidělení svazků je nezbytné z několika důvodu. V případě databáze zajistí svazek perzistentní uložení dat na hostiteli. V rámci aplikace Collector se ale o persistenci nejedná. Svazek je zde přidělen pro umožnění čtení pseudosouborového systému Cgroups a proc. Díky tomu může aplikace Collector sledovat všechny běžící kontejnery na hostiteli.

5.7.1 Použité Docker image

Pro kompilaci a běh bylo využito několika Docker image:

- **alpine:latest** — je základní verze Alpine image. Obsahuje v základu vše potřebné pro běh C / C++ programů. Pro kompilaci s tímto image je nutné přidat software cmake. Tento image se používá při kompilaci a běhu C++ aplikace Collector.

- **maven:3.8.4-openjdk-17-slim** — je image vhodný pro kompilaci Java programů obsahující JDK 17 a Maven. Právě díky nástroji Maven můžeme Java Spring aplikace zkompilovat. Zmíněný image se používá pro kompilaci Java Spring aplikace Monitor.
- **openjdk:17-jdk-alpine** — je image s Alpine verzí linuxu a JDK 17. JDK implementuje Java SE Platform 17 obsahující virtuální stroj, vývojové nástroje, technologie nasazení a další knihovny díky kterým je možné spustit Java programy. Image je využit v případě běhu Java Spring aplikace Monitor.
- **mcr.microsoft.com/dotnet/sdk:5.0** — je Microsoft image s `dotnet` kompilátorem. Tento image se používá pro kompilaci Blazor
- **nginx:alpine** — je Alpine image s reverse proxy serverem NGI X pro HTTP, HTTPS, SMTP, POP3 a IMAP protokoly. Je vhodný pro běh webových aplikací a proto je použit pro běh Blazor aplikace Viewer.

5.8 Pokročilá konfigurace systému

Celý monitorovací systém je založen na co nejjednodušší konfiguraci a není tak nutné nějakým způsobem zasahovat do konfiguračních souborů. Systém je převážně konfigurovatelný prostřednictvím webového rozhraní, jak již bylo popsáno v sekci 5.6.1. Existují ale konfigurace, které lze provádět pouze za pomocí konfiguračních souborů. Jedná se o nastavení, které není nutné při běžném užívání, ale slouží pro pokročilou konfiguraci systému.

Mimo souborů pro konfiguraci propojení jednotlivých rozhraní, je v aplikaci Collector soubor nesoucí název config.ini. Jedná se o konfigurační soubor, který se načítá při spuštění monitorovacího sytému. Modifikace při běhu tak neaplikuje požadované změny přímo, ale až při restartu. Tento soubor je ve známém formátu ini a obsahuje několik nastavení z nichž za zmínku stojí sekce Zpoždění a Ban list.

- **Zpoždění** — Tato sekce nastavení obsahuje dva parametry metric_delay a explore_delay s hodnotou zpoždění. Hodnota metric_delay odpovídá intervalu v milisekundách, po kterém se přečtou metriky všech kontejnerů. Hodnota parametru explore_delay odpovídá intervalu, po kterém se zkontrolují kontejnery pro případ spuštění nového, který je zapotřebí sledovat. Jednotky jsou obdobné jako v případě prvního parametru v milisekundách.
- **Ban list** — V této sekci nalezneme seznam jmen kontejnerů, které jsou vyčleněny z procesu monitorování. Uživatel zde může přidat kontejner, který nevyžaduje sledovat.

Kapitola 6

Testování

Testování je jedním ze zásadních faktorů pro správně fungující systém. Jedná se o nedílnou součást vývoje každého softwaru. Pomáhá nejen v implementační části, ale současně v případě budoucího rozšíření pro kontrolu, zde vše funguje jak má.

V následujících sekcích kapitoly Testování se popisuje způsob, jakým byl monitorovací systém testován. Testovány byly jednotlivé části/aplikace nejen samostatně pomocí Unit a integračních testů, ale také jako celek pomocí testů systémových. Pokud není uvedeno jinak, jsou informace čerpány ze zdroje [62].

6.1 Testovací framework

Před testováním je často vhodné zvážit výhody některého z testovacích frameworků a jeho následné použití při automatizovaném testování. Framework určený pro testování je sada procesů, pravidel, nástrojů a osvědčených postupů používaných pro tvorbu a navrhování testovacích případů. Testovací případ je soubor podmínek, za kterých tester určí, zda systém nebo jedna z jeho částí funguje tak, jak bylo původně zamýšleno. Takovýto framework lze následně použit pro automatizované testování softwaru. [57]

Pozitivem testovacích frameworků je hned několik. Za nejdůležitější pozitivum se dá považovat poskytnutí prostředku pro efektivní testování, díky kterému se proces testování značně zrychlí. Za zmínku také stojí poskytování postupů, díky kterým je možné dosáhnout určité uniformnosti testů, jenž vedou k snadnější údržbě. [57]

6.1.1 Testovací framework pro Java aplikace

Testovacích frameworků pro testování Java aplikací je na trhu hned několik. Jedním z velice známých a běžně používaných řešení pro testování je JUnit. Díky jeho rozšířenosti a časté integraci s různými IDE je vhodnou volbou pro testování aplikace Monitor. [58]

JUnit je open-source framework pro testování, který používají především vývojáři softwaru k jednotkovému testování. Nicméně lze využit rovněž pro složitější integrační testy. Nejnovější verzí JUnit je verze 5, která vyžaduje pro chod Javu 8. Zmíněná verze také podporuje spuštění testů starších verzích JUnit. Tento framework zaujímá podstatné postavení na trhu již řadu let a není tak divu, že má velice propracovanou dokumentaci. [58]

6.1.2 Testovací framework pro C++ aplikace

Při výběru testovacího frameworku pro C++ již není cesta tak přímočará, jako v případě testování jazyka Javy. Na výběr je spousta možností od open-source řešení až po proprietární nástroje. V rámci známých nástrojů pro testování můžeme vybírat například z řešení jako Google Test, Cppunit, Catch2 nebo Boost Test. [59]

Výběr technologie závisí zcela na očekávání, které se od frameworku požaduje. Například signifikantním rozdílem frameworku Cppunit je jeho podobnost s JUnit frameworkem. Výhodou Catch je zase rychlost a snadné použití. V případě Boost Test je výhoda v jeho nasazení, jelikož je součástí knihovny Boost. Velice populárním je bezpochyby Google test s velkou podporou řady platforem a širokou škálou užitečných funkcí. [59]

Ač byla volba této technologie nelehká, pro testování aplikace Collector byl zvolen framework Google Test a to hned z několika důvodů. Google Test je open-source řešení, které je v dnešní době jedno z nejpopulárnějších a poskytuje vhodné prostředí pro snadné testování C++ projektů. Je vhodný především na unit testy, ale zároveň lze použít i na testy integrační. Velkou výhodou je obsah Google Mock knihovny v samotném základu, jelikož většina ostatních řešení mock objekty neposkytuje. [59]

6.1.3 Google Test framework

Veškeré následující informace v této sekci Google Test frameworku a způsobu jeho použití byly získaný z Google Test dokumentace [60]. V případě použití Google Test frameworku je nejprve nutné nakonfigurovat prostředí tak, aby bylo možné testy spustit. Pro konfiguraci nejprve vytvoříme nový textový soubor CMakeLists.txt určený pro testování. Tento soubor bude obsahovat cesty k souborům se zdrojovými kódy a také použité knihovny. Následně soubor CMakeList.txt použije CMake pro sestavení projektu. Příklad obsahu zmocňovaného souboru lze pozorovat na ukázce viz 9. Tímto je veškerá nutná konfigurace nastavena a lze dále psát testy.

Každý soubor obsahující testy musí importovat knihovnu gtest/gtest.h, které obsahuje potřebná makra pro testování. Dále soubor obsahuje již samotné testovací případy. Knihovna gtest obsahuje několik druhů testů. Jedním z použitých je test pomocí makra `TEST`, jenž je určen pro testování statických nebo globálních funkcí. Druhým použitým testem je `TEST_F`, který umožňuje

```
project(Google_tests)
add_subdirectory(lib)
include_directories(${gtest_SOURCE_DIR}/include ${gtest_SOURCE_DIR})
add_executable(project_run ....)
target_link_libraries(project_run gtest gtest_main qmock)
```

Výpis kódu 9: Příklad CMakeLists.txt

přijmout objekt s předdefinovanou strukturou pro testování (Fixture). Tato struktura je užitečná zejména v případech, kdy ve více testech používáme stejné konfigurace. Strukturu Fixture můžeme vidět na příkladu 10.

```
class ContainerExplorerFixture : public ::testing::Test {
protected:
    virtual void SetUp()
    {
        executor = std::make_shared<MockDockerExecutor>();
        parser = shared_ptr<parser::DockerParser>(new parser::DockerAPIParser);
        pathGenerator = shared_ptr<PathGenerator>(new LinuxPathGenerator());
        controller = std::make_shared<MockDBController>();

        string data = "XML...."
        string pid = "XML...";
        EXPECT_CALL(*executor, getContainers()).WillOnce(Return(data));
        EXPECT_CALL(*executor, getPid(testing::_)).WillRepeatedly(Return(pid));
    }
    std::shared_ptr<MockDockerExecutor> executor;
    std::shared_ptr<parser::DockerParser> parser;
    std::shared_ptr<PathGenerator> pathGenerator;
    std::shared_ptr<MockDBController> controller;
};
```

Výpis kódu 10: Příklad s Fixture

Pokud je nutné použít Mock objekt pro simulaci, je zapotřebí importovat knihovnu gmock/gmock.g. Pro vytvoření Mock objektu stačí definovat třídu, která bude dědit od simulovaného objektu. Následně všechny metody, které budou na Mock objektu volány, je nutné definovat pomocí marka `MOCK_METHOD`, jak můžeme vidět v ukázce 11. Nad takto definovanými funkcemi lze použít v testovacím případu další makra `EXPECT_CALL`, které mohou fungovat několika způsoby a to jako assertion nebo pouhé definice, jak má funkce odpovídat. Příklad použití makra `EXPECT_CALL` je možné pozorovat na obrázku 10, kde je v prvním případě na první volání funkce `getContainers()` odpovězeno hodnotou odpovídající proměnné data.

6.1.4 JUnit testovací framework

Informace a použití JUnit frameworku byly získány z dokumentace JUnit 5 [61]. Framework JUnit zjednodušuje samotné testování. Pro jeho použití je

```
class MockDockerExecutor : public DockerExecutor {
public:
    MOCK_METHOD(string, getPid, (const string & containerID), (override));
    MOCK_METHOD(string, getContainers, (), (override));
};
```

Výpis kódu 11: Příklad definice Mock objektu

zapotřebí přidat pro nástroj Maven do pom.xml konfiguračním souboru jednu závislost `spring-boot-starter-test`. Dále lze již využívat anotací od JUnit a spring frameworku pro testování aplikací.

Pro specifikaci třídy, která bude obsahovat testovací případy, mimo volitelných konfiguračních anotací, je nutné uvést anotaci `@SpringBootTest`. Takto uvozená třída již může obsahovat testy, které lze spouštět jednotlivě nebo jako celek. Testovací případ může být uveden několika různými anotacemi. Při testování aplikace Monitor byla využita základní anotace `@Test` a anotace `@ParameterizedTest` jenž dovoluje test parametrizovat. Specifikovat takové parametry je možné pomocí funkce ve tvaru viz 12. Dále v testech pro kontrolu výsledku je možné využít funkce z balíčku `Assertions`.

```
static Stream<Arguments> paramProvider() {
        return Stream.of(
                arguments(true, 10,3000,Trigger.ABOVE,Threshold.AVERAGE,15),
                arguments(true, 9,1500,Trigger.ABOVE,Threshold.AVERAGE,10),
                arguments(false, 10,1500,Trigger.ABOVE,Threshold.AVERAGE,10),
        );
    }
```

Výpis kódu 12: Příklad definice funkce poskytující parametry pro testy

V případě, kdy je zapotřebí simulace objektu, lze i zde využít výhod Mock objektů. Mokovat (simulovat) lze nejen třídy, ale také Java Beans. Simulovat Bean lze pomocí anotace `@MockBean`, jenž přepíše samotnou Bean. V testech je pak možné určit její konkrétní chování pomocí `Mockito.when`.

6.1.4.1 Konfigurace H2 databáze

V případech, kdy testovaná aplikace využívá ke svému správnému fungování databázový stroj, je často nutné tento stroj zastoupit pro potřeby testování. Často se za tímto účelem zastoupení používají takzvané *in-memory* databáze. Jednou z těchto databází je H2, která byla použita při testování aplikace Monitor.

Mimo přidání závislosti do pom.xml konfiguračního souboru je databázi nutné pro potřeby testování také nakonfigurovat. Za tímto účelem je možné specifikovat konfigurační soubor `.properties` se specifikací *datasource* (konfigurace databáze). Následně pro využití právě tohoto nově vytvořeného souboru je nutné přidání anotace `@TestPropertySource` nad třídu s testy. Díky

tomu je zřejmé, jaký konfigurační soubor se má při testech použít. Anotace pro testování a specifikaci nového konfiguračního souboru lze pozorovat na ukázce s kódem 13.

```
@TestPropertySource(locations= "classpath:defaultTest.properties")
@SpringBootTest
@ContextConfiguration(classes = DatabaseTestConfiguration.class)
@Sql("classpath:schema.sql")
@Sql("classpath:import.sql")
public class TestClass {...}
```

Výpis kódu 13: Příklad anotací pro dodatečnou konfiguraci testů

Ve zmíněné ukázce je rovněž možné pozorovat dodatečné nastavení uvedené anotací `@ContextConfiguration`. Díky této anotaci je možné definovat třídu, která bude obsahovat například předefinování vybraných Java Bean.

6.2 Unit testy

Unit nebo také jednotkové testy jsou používány za účelem testování jednotlivých funkcionalit. Za testování funkcionality můžeme považovat například test při kterém testujeme jednotlivé funkce dané třídy. Cílem je tedy izolované testování každé části/funkce programu a dokázání, že jednotlivé části aplikace fundují dle předpokladů. Pro unit testování je typické automatické provádění na základě programátorem definovaných testů. Často se pro zjednodušení psaní takových to testů využívají různé testovací frameworky specifické pro daný programovací jazyk.

Unit testy byly použity jak pro testování funkcionalit aplikace Collector, tak aplikace Monitor. Tímto způsobem testování nebyly pokryty veškeré části (funkce) aplikací ale pouze zásadní části, u kterých je vetší pravděpodobnost výskytu chyby.

V přisadě aplikace Collector byly testovány třídy poskytující parsovací funkcionality. Konkrétně se jednalo o funkcionality pro parsování pseudosouborů s metrikami nebo také parsování příchozích odpovědí vycházejících z komunikace s ostatními aplikacemi. Další testovanou částí programu byly exportery, které řeší v aplikaci správný odesílací formát. Příklad takového testu zabývajícího se exporty dat pro Influx line protokol můžeme vidět na ukázce 14.

Hlavní částí testování unit v případě aplikace Monitor zaujímaly testy repository. Náplní těchto testů je zjistit, zda se záznamy namapují správně na objekty. Právě v tomto případě byla využita zmíněná databáze H2 se strukturou obdobnou databází QuestDB naplněná testovacími daty. Příklad testování kontejner repository 15.

```
TEST(InflixLineProtocolExporterTest, Syntax) {
    InfluxLineProtocolExporter exporter = InfluxLineProtocolExporter();
    exporter.addTableName("CPU");
    exporter.addTag("id","gm");
    exporter.addAttribute("kernel",10);
    exporter.addAttribute("user",10);
    exporter.addAttribute("test","test");
    exporter.addTimestamp("123456789");
    EXPECT_EQ("CPU,id=gm kernel=10i,user=10i,test=\"test\" 123456789\n",
                exporter.exportData());
}
```

Výpis kódu 14: Test Influx line exporteru

```
@Test
public void containerParsingTest(){
    List<Container> containers = containerRepository.findAll();
    assertThat(containers.size()).isEqualTo(3);
    assertThat(containers.get(0).getDockerID()).isEqualTo("0264d2962ddb8f8");
    assertThat(containers.get(0).getId()).isEqualTo("data_collector");
    assertThat(containers.get(0).getImage()).isEqualTo("data_collector");
    assertThat(containers.get(0).getName()).isEqualTo("/collector,");
}
```

Výpis kódu 15: Příklad repository testů

6.3 Integrační testy

Integrační testy mají za cíl otestovat aplikaci z pohledu spolupráce komponent. Testují, zda komponenty mezi sebou komunikují správně. Zabývají se tak testováním většího celku aplikace a ne specifickou funkcionalitou jako je to v případě Unit testů. Testování probíhá většinou v podobě simulace reálného prostředí, kterému budou komponenty v ostré verzi vystaveny. Provádějí se většinou v automatické podobě za pomocí některého testovacího frameworku.

Stejně jako unit testy byly použity i integrační testy na obou aplikacích Collector i Monitor v podobě automatických testů. Vždy byly testovány hlavní moduly a jejich integrace s okolím.

Aplikacie Collector je sestavena ze dvou hlavních modulů pro čtení metrik a objevování kontejnerů. Oba tyto moduly byly podrobeny důkladným integračním testům, jenž pokrývají reálné situace, do kterých se mohou v průběhu chodu dostat. Jeden z takových to scénářů, kdy Docker spustil několik nových kontejnerů můžeme pozorovat na ukázce 16.

V případě aplikace Monitor jsou integrační testy zaměřeny primárně na modul zabývající se monitorováním metrik a dále na notifikace v případě naplnění některého pravidla. Jedná se o složitý proces, ve kterém je největší pravděpodobnost na výskyt chyby a proto je vhodné jej důkladně otestovat. Do testovacích případů byly začleněny testy zabývající se správným časováním a také testy obou notifikačních módů Threashold i Change.

```
TEST_F(ContainerExplorerFixture, exploreNewWithExistingCTest) {
    EXPECT_CALL(*controller, initContainer(testing::_)).Times(AtLeast(3));
    ContainerExplorer explorer =
        ContainerExplorer(executor,parser,pathGenerator,blackList);
    vector<Container> containers =
        {Container("1d918f6bc1b",{},10,"questdb/questdb:latest")};

    explorer.exploreNew(containers,controller);
    EXPECT_EQ(4,containers.size());
    EXPECT_EQ(10,containers[0].getPid());
    EXPECT_EQ("questdb/questdb:latest",containers[0].getImage());
    EXPECT_LE(1,containers[0].getMetricsPath().size());
    EXPECT_EQ("openzipkin/zipkin",containers[1].getImage());
    EXPECT_EQ("redis",containers[2].getImage());
    EXPECT_EQ("daprio/dapr",containers[3].getImage());
}
```

Výpis kódu 16: Integrační test pro případ vzniku nových kontejnerů

6.4 Systémové testy

Poslední testovací vrstvou v navrhovaném systému jsou systémové testy. Mají za cíl otestovat, zda systém funguje jako celek dle zamýšlení. Zaměřují se zejména na integraci s ostatními systémy, a tedy se testuje celá aplikace jako celek. Jedná se o testy typicky prováděné jako Black box (zaměřujeme se na vstupy a výstupy bez znalosti implementace). Lze je provádět jak automaticky pomocí nástrojů tak i manuálně. V této práci se využilo manuálního testování.

Prověřena byla nejen integrace, ale také veškeré požadované funkčnosti systému. Pod pojmem integrace je v tomto případě myšlena integrace systému z pohledu jeho jednotlivých částí i integrace s ostatními aplikacemi jako Docker, Slack nebo pseudo-soborovými systémy. Testování probíhalo nejprve prostřednictvím samotného REST API a následně i pomocí webového grafického rozhraní. Vedení manuálních testů postupovalo ve formě *testcase* (testovací případ). Jedná se o testování aplikace dle předem připravených sad instrukcí. Jedním z takových to případů je například: *Zkontroluj výsledek při zadání špatného uživatelského jména při přihlášení do systému.*

Testování dle předem připravených případů nebyl jediný způsob manuálního testování celého systému. Dalším použitým způsobem jsou testy zvané *Exploratory*. Jejich rozdíl oproti testcase jsou chybějící instrukce. Tester tak postupuje dle vlastního uvážení. Volná ruka testera má za následek možnost objevení i skrytých závad vlivem neobvyklých kroků.

Kapitola 7

Nasazení informačního systému

Jedním z cílů při návrhu monitorovacího systému bylo snadné nasazení a konfigurace. Výsledek tohoto záměru je výrazně viditelný v této kapitole, jelikož nasazení v případě splnění nezbytných požadavků je otázkou jednoho příkazu a dá se tedy považovat za snadné.

V následujících sekcích tato kapitola uvede požadavky nutné pro správný chod monitorovacího systému. Dále v závěru uvede podrobné instrukce jak samotný systém nasazení na hostitelské zařízení.

7.1 Požadavky

Monitorovací systém vyžaduje určitě požadavky pro jeho chod. Mimo samozřejmého požadavku v podobě instalované verze Docker Enginu je nutné zajistit několik dalších. Jedněmi z nich je přistup k internetovému připojení pro stažení nejen monitorovacího systému, ale také nutných referencí, knihoven a Docker images. Dalším požadavkem je nástroj Docker-compose, jelikož není instalován současně s Dockerem a je ho tak nezbytné dodatečně doinstalovat.

7.1.1 Postačující požadavky

Vhodným hostitelským operačním systémem je Linuxová distribuce Ubuntu, jelikož obsahuje mnohé nezbytné požadavky již v základu. Monitorovací systém se ale na specifickou distribuci Linuxu neváže. Je tak možné využít i jinou. V takovém případě je nutné dodržet veškeré požadavky uvedené dále 7.1.2. Při splnění následujících požadavků bude monitorovací systém pracovat správně.

- Volné porty 80 a 8080 (8080 pro REST API a 80 pro webovou aplikaci)
- Docker Engine 20.10.7
- Docker Compose 1.29.2
- Linux Ubuntu 20.04 (pozor Ubuntu 21 již má v základu Cgroup v2)

7.1.2 Nezbytné požadavky

V případě kdy monitorovací řešení nebude nasazeno na operačním systému Ubuntu, je zapotřebí zkontrolovat následující požadavky.

- Právo Read nad Cgroups pseudo-souborovým systémem
- Umístění Cgroups v `/sys/fs/cgroups`
- cgroup v1
- Právo Read nad proc pseudo-souborovým systémem
- Docker API verze 1.25 a novější
- Docker Engine 19.03.0+ pro 3.9 `compose.yml`
- Volné porty 80 a 8080
- Docker socket umístěn na `/var/run/docker.sock`
- Docker Compose 1.29.2

Zmíněné požadavky jsou nutné pro správný chod a v případě porušení jakéhokoli z nich je jisté selhání některé části monitorovacího systému. Jedinou výjimkou je umístění Docker socketu a cest k pseudo-souborovým systémům. V případě nutnosti lze tyto cesty upravit přímo v kódu před samotným spuštěním.

7.2 Nasazení

Veškeré nutné soubory pro spuštění monitorovacího systému jsou uloženy na github cloudu. Stačí je stáhnout pomocí příkazu nebo manuálně. Umístění je zcela na volbě uživatele.

Po stažení je nezbytné přejít do složky gama_monitor a spustit monitorovací systém pomocí příkazové řádky příkazem `docker-compose up -d`. Parametr `-d` slouží pro spuštění na pozadí. Po několika vteřinách již systém začne automaticky sledovat metriky. V případě dodatečného nastavení je nutné jej provést před samotným startem 17.

7.2.1 Instalace Docker Compose

V případě chybějícího nástroje Docker Compose je možná jeho do instalace pomocí následujících příkazů 17.

```
sudo curl -L "https://github.com/docker/compose/releases/download/
        1.26.0/docker-compose-$(uname -s)-$(uname -m)"
            -o /usr/local/bin/docker-compose
sudo mv /usr/local/bin/docker-compose /usr/bin/docker-compose
sudo chmod +x /usr/bin/docker-compose
```

Výpis kódu 17: Příkazy pro instalaci nástroje Docker Compose

7.2.2 Změna Cgroups verze

Některé operační systémy používají Cgrouvs verze 2. Tato verze mění oproti verzi 1 celkovou hierarchii, na kterou není monitorovací systém připraven. V případě, že hostitel používá zmíněnou verzi V2 je možné ji změnit na verzi V1. Pro provedení této akce postačuje úprava parametru v kernelu. Pro změnu na verzi V1 je možné aplikovat na hostiteli následující Bash příkaz 18. Změna není okamžitá, ale projeví se až po restartu hostitelského zařízení. Následně je již používán Cgroups verze V1 a je tak možné nasadit monitorovací systém.

```
sudo grubby --update-kernel=ALL --args="systemd.unified_cgroup_hierarchy=0"
```

Výpis kódu 18: Příkazy pro změnu verze Cgroups

Kapitola 8

Rozšíření a přínosy

Navržený monitorovací systém je velice komplexní program a v jeho rozšiřování o další funkcionality se téměř meze nekladou. Vždy je možné systém vylepšit nebo dodat další funkcionalitu na základě specifického přání uživatelů.

Následující kapitola tak seznamuje s možnostmi budoucího rozšíření implementovaného softwaru společně s dosud zjištěnými informacemi k dané problematice. Dále pojednává o přínosech, jaké monitorovací systém přináší společně s rozborem, pro jaké uživatele je vhodný. Na závěr kapitoly, je monitorovací systém porovnán s některými již zmíněnými řešeními, jak ve stejné kategorii neplacených řešeních, tak i mimo ni.

8.1 Rozšíření

Rozšiřování je možné téměř v každém směru a v každé části aplikace. Systém má tak velký potenciál na další rozšíření od podpory a integrace dalších platforem, zaznamenávání dalších metrik, nebo zlepšení analýzy metrik. Je tak velice obtížné zmínit veškeré možné rozšíření, jak se může dál monitorovací aplikace vyvíjet a proto budou nastíněna pouze vybraná zajímavá rozšíření. U některých rozšíření bude nastíněn i možný scénář, jak jej implementovat s dosud zjištěnými znalostmi o dané problematice.

8.1.1 Rozšíření pro podporu Docker Swarm

Monitorovací systém je již v základu navrhován s myšlenkou použití i pro monitorování kontejnerů běžících v Docker Swarm módu. Proto je při návrhu a implementaci striktně oddělena aplikace Collector s možností běhu ve více instancích bez ovlivnění celého systému. Díky tomu může být nasazena na každého hostitele zvlášť a sbírat tak jeho metriky.

Nasadit aplikaci Collector na všechny hostitele lze manuálně, v takovém případě by řešení mělo fungovat v podobě, v jaké se nachází. Tento postup ale není příliš uživatelsky vhodný. Nejen že se nasazení monitorovacího systému

stává složité, ale zároveň v případě výpadku hostitele nebo vzniku nového, je nutné se startem hostitele současně spustit Collector aplikaci.

Tento proces se dá automatizovat pomocí specifikace `deploy` v compose s hodnotou `mode: global`. Tato specifikace dle Docker dokumentace zajistí spuštění zvoleného kontejneru právě jednou na každém Swarm node. Funkčnost této metody ale není řádně otestována a je nezbytné jí dále detailněji prozkoumat, proto je zařazena jako možné rozšíření.

8.1.2 Rozšířen integrací notifikačních agentů

V této chvíli monitorovací systém podporuje pouze jednoho notifikačního agenta a tím je Slack. Je tedy na místě uvažovat v budoucnu o rozšíření tímto směrem například ve formě zasílání emailových zpráv nebo dalších agentů podobných jako je Slack. Příkladem takovýchto agentů může být například Microfost Teams, Jira nebo Hangouts.

Implementace z hlediska notifikační části je přizpůsobena právě pro vznik dalších agentů. Pro jejich přidání stačí napsat implementaci a zaregistrovat tohoto agenta jako pozorovatele (observer).

8.1.3 Sledování kontejnerů i jiných kontejnerizačních nástrojů

Jako další možné rozšíření monitorovacího systému je expanze mimo sledování pouze Docker kontejnerů. Nabízí se možnost integrace i s dalšími nástroji pro kontejnerizaci, jako je například velice známá platforma Kubernetes.

V případě kdy kontejnerizační nástroj nebo samotně sledovaná aplikace využívá Cgroups, je rozšíření pro sledování metrik velice jednoduché, jelikož lze využít téměř celou aplikaci Collector a zbytek systému. Nutným rozšířením je pouze servis, jenž získá informace potřebné pro vyhledávání metrik v pseudo-souborovém systému Cgroups.

V opačném případě, kdy není možné využít Cgroups, je nutné rozmyslet a implementovat jiný způsob získávání metrik a tedy přepsat téměř celou aplikaci Collector. V takovém případě je na zvážení, jestli nevyužít jiné řešení. Jelikož zmíněné rozšíření by znamenalo velké změny.

8.1.4 Sledování dalších metrik

Z hlediska sledování metrik, monitorovací systém sleduje metriky z Cgroups a proc pseudo-souborových systémů. Jedná se tedy převážně o metriky využití paměti, disku a procesoru. Lze tedy uvažovat o rozšíření škály sledovaných metrik o další. Příkladem takových to metrik jsou například různé síťové latence. V případy kdy dojde k výpadku, ať už na lokální síti nebo úplnému výpadku připojení internetu, lze tyto metriky využít k odhalení problému.

8.1.5 Rozšířená analýza metrik

Podstatnou částí každého monitorovacího systému je jeho vizuální stránka zobrazující naměřené metriky. V tomto ohledu navrhovaný systém poskytuje pouze základní pohled na data, který nelze příliš přizpůsobovat. Dalším rozšířením, které může přinést velký přínos je tedy možnost vytvoření vlastního dashboardu s potřebnými informacemi. Díky tomu může uživatel sledovat pouze to, co sám uzná za podstatné a není rozptylován ostatními informacemi.

Současně s vizualizací metrik se také váže forma, v jaké jsou zobrazeny. Často je vhodné, ba i žádané, data nějakým způsobem agregovat či vizualizovat chování za uplynulý čas. Příkladem může být zobrazení trendů metriky přímo v grafech společně s živými daty. Uživatel tak má hned přehled o tom, jestli se chování nezměnilo.

8.2 Přínosy

Samotný systém pro monitorování má přínos jako celek, tak i samostatně jako jednotlivé části. V případě celku se jedná o velice rychle nasaditelné monitorovací řešení schopné monitorovat metriky a upozorňovat uživatele na nežádoucí chování. Výhodou je rovněž vysoká prioritizace nízkých nákladů na provoz. Díky čemuž má systém velké spektrum využití, a to i v případě sledování jen několika kontejnerů. Pro využití plného potenciálu systému, je ale nutné dále prozkoumat již zmíněné rozšíření pro usnadnění nasazení na Docker Swarm.

Systém má také přínos i z pohledu jeho samostatných částí. Konkrétně stojí za zmínku aplikace Collectro, kterou lze využít, díky její samostatnosti, k ukládání metrik přímo do souborů. Lze tak sledovat kontejnery a jejich metriky živě s naprosto zanedbatelnými náklady a s ponecháním postrocesingu metrik na uživateli.

Přínosu aplikace Collector, v podobě možnosti běhu samostatně s ukládáním živých metrik do souboru, bylo již využito pro potřeby analýzy v této práci. Konkrétně se jednalo o rozhodnutí pro volbu databázové platformy. Modifikovaná aplikace Collector v tomto případě sledovala zvolený kontejner s běžící databází. V průběhu tohoto sledování byly databáze zatěžovány k zjištění jejích reálných nároků. Výsledek sledování byl pak použit v grafech a tabulkách 4.2.4 a tedy následně pro rozhodnutí, která databázová platforma je nejvhodnější.

Neopomenutelným přínosem je také psaná část práce pro kohokoli, kdo hledá informace o sledování metrik kontejnerů. Muže tak využít tuto práci s možnostmi a konkrétními příklady jak metriky sledovat společně s formáty, ve kterých jsou metriky uloženy.

8.2.1 Zátěž

Jedním z přínosů systému je nízké zatížení hostitele. Již při návrhu a implementaci bylo podrobně zkoumáno, jak docílit monitorovacího systému s nízkými nároky jak z hlediska velikosti zabíraného místa na disku, tak i z hlediska požadavků na hardware při monitorování. Tento záměr byl docílen primárně díky využití vhodných technologií. Jednou z nich je volba technologie získávání metrik pomocí čtení pseudo-souborů pomocí nízkoúrovňového jazyka C a rovněž vhodné volbě databáze specializované na časové záznamy. K nízkým nárokům také přispěla technologie Blazor Webassembly, díky které je možné využít výkonu uživatelského zařízení namísto serveru.

8.2.2 Modularita

Modularita je velkým přínosem z hlediska dalšího rozšiřování systému. Díky ní je možné bez zbytečného přepisování rozšířit systém o další funkcionality, nebo změnit stávající bez velkého zásahu do již funkčního kódu.

V monitorovacích a potažmo i jiných softwarech je časté software přizpůsobovat potřebám klienta či uživatelům. Je tedy vhodné mít systém modulární pro rychle a snadné rozšíření. Tohoto cíle je dosaženo i v případě navrhovaného monitorovacího systému nejen díky modulům, ale také na základě rozdělení do několika oddělených aplikací. Díky tomu je vhodným základem pro další rozšiřování k obrazu specifických požadavků konkrétních uživatelů.

8.2.3 Kdo systém využije

Jako potencionálního uživatele si můžeme představit kdokoli, kdo potřebuje vizuálně sledovat či monitorovat metriky kontejnerů. Řešení se hodí nejen na sledování live dat, ale současně na dlouhodobé analýzy díky možnosti sledování metrik již od samotného spuštění kontejneru s limitujícím faktorem pouze v hostitelově velikosti disku.

Monitorovací systém není příliš vhodný pro obrovské společnosti vyžadující komplexní řešení a nepřetržitou podporu v případě potíží. Takovým to společnostem často nejde o rychlost nasazení nebo efektivitu, ale spíše hledí na propracovanost a funkcionality. Proto systém nelze porovnávat s komerčními řešeními, jenž přicházejí se spoustou funkcionalit a podporou při nasazení.

Proto se řešení zaměřuje spíše na menší uživatele/klienty vyžadující snadné a rychle nasazení. Ideální je pro použití v případě, kdy sledujeme kontejnery a chceme být upozorněni například na výpadek služby nebo započetí swapování poměti a zároveň nechceme obětovat přílišný výkon hostitele. Řešení je také vhodné například pro testování nově nasazených kontejnerů a sledování jejich metrik v případě různého zatížení (benchmarků).

8.3 Porovnání s existujícími řešeními

Pro porovnání byly vybrány již zmíněné řešení z analýzy 3.3. Jedno komerční placené a několik bezplatných řešení. Jelikož je navrhované řešení myšleno jako open-source projekt je realistické jej srovnávat s řešeními stejné úrovně, tudíž s bezplatnými platformami.

8.3.1 Porovnání s kombinací cAdvisor, Prometheus a Grafana

V případě open-source řešeních není příliš mnoho komplexních monitorovacích systémů s možností zobrazení a notifikací společně s dlouhodobým uchovávání metrik. Pro tento cíl se ve většině případů používá kombinace několika řešení dohromady jako cAdvisor, Prometheus a Grafana.

Spoluprací těchto softwarů dohromady dosáhneme veškeré potřebné funkcionality, které nabízí implementované řešení, na úkor několika věcí. Jedná se primárně o větší nároky na provoz společně s velice složitou konfigurací a nasazením. Nasazení lze zjednodušit pomocí využití předem připravených docker-compose konfiguračních souborů od komunity s již nakonfigurovanými Docker images. Tento krok ale nezjednoduší konfiguraci ani použití. Stále musíme pro konfiguraci používat několik rozhraní.

Nevýhody využití kombinace:

- náročnost na provoz,
- komplikovanější nasazení,
- nastavování prostřednictvím několika rozhraní,
- složité nastavení notifikací (textový soubor),
- chybějící počáteční nastavení vizualizace metrik.

Výhody využití kombinace:

- možnost vytvoření vlastního dashboardu,
- lepší vizualizace metrik,
- širší možnosti sledování (nejen kontejnerů),
- integrace s více systémy,
- možnost ukládání grafů.

8.3.2 Porovnání s cAdvisor

Pokud budeme implementované řešení monitorovacího systému porovnávat pouze s cAdvisor systémem, je zásadní podotknout několik věcí. cAdvisor se soustředí na jednoho hostitele a jeho metriky. Nesleduje pouze kontejnery, ale veškeré běžící procesy na hostiteli. Díky tomu nemá bližší informace o kontejnerech. cAdvisor neumožňuje data ukládat po delší dobu a znázorňuje tak pouze live data. Má tak o něco jiné cíle, než implementovaný systém.

V případě porovnávání vytíženosti cAdvisor nezatěžuje příliš hostitele. Využívá přibližně 2–3 % celkového vytížení CPU a 140MB paměti. V porovnání s celým navrhovaným systémem je jeho zatíženi zanedbatelné. Pokud budeme srovnávat vytížení ne s celým systémem, ale pouze s jednou jeho částí zajišťující čtení metrik (aplikace Collector) výsledek dopadne následovně. Aplikace Collector využívá ke svému chodu průměrně 1–1,5 % CPU a 5MB paměti. Samotné část je tedy šetrnější na využití zdrojů. Nejedná se ale o nějak zásadní hodnoty. Testovaní a získání metrik o vytížení bylo provedeno na hostiteli s konfigurací viz 4.2.4.1.

Nevýhody cAdvisor:

- nemožnost notifikací,
- neumožňuje dlouhodobé ukládání,
- konfigurace pomocí XML struktury,
- nemá přehledný dashboard.

Výhody cAdvisor:

- podporuje monitorování nejen kontejnerů ale i procesů,
- integrace s více systémy,
- spousta návodů a rad,
- menší náročnost na zdroje.

8.3.3 Porovnání s DataDog

Následující srovnání s monitorovacím řešením DataDog není příliš realistické a nelze tak brát jako konkurenční. DataDog je closed-source placené řešení jenž dokáže sledovat nejen kontejnery, ale aplikace jako takové. Díky tomu může poskytovat i podrobnější metriky než implementovaný systém. Co se týká uživatelského rozhraní a počtu funkcionalit je DataDog rozhodně bezkonkurenční. V čem získává implementovaný systém navrch je snadnost a rychlost nasazení. Vzhledem k možnostem a rozmanitým integracím je nasazení DataDog o něco složitější. Tento fakt ale z části maže podpora od společnosti

DataDog.

Nevýhody DataDog:

- jedná se o placené řešení,
- komplikované nastavení a integrace,
- výkonnostně náročnější.

Výhody DataDog:

- možnost sledování i aplikací,
- detailní metriky pro konkretní platformy,
- širší integrace se servery pro odesílání notifikace,
- uživatelsky upravitelný dashboard,
- zákaznická podpora.

Závěr

Cílem této diplomové práce bylo analyzovat, navrhnout a následně i implementovat funkční nástroj pro sledování základních metrik Docker kontejnerů s důrazem na jeho snadnost nasazení a konfiguraci. Nástroj musí podporovat jak samotné monitorování a ukládání metrik, tak současně konfiguraci a zasílání upozornění na základě specifikovaných pravidel.

Součástí analýzy bylo seznámení s problematikou, rozbor možností sledování metrik a rovněž analýza existujících řešení jak v open-source sféře, tak i komerční. Díky této analýze bylo možné získat lepší představu o problematice společně s objevením dalších nezbytných požadavků na finální systém. Při analýze monitorovacích řešení byla také zjištěna celá řada softwarů zabývající se právě sledováním metrik. Spousta z nich je ale složitá jak pro nasazení, ta i konfiguraci. Proto navrhovaný systém klade velký důraz právě na tento faktor.

Velkou část analýzy, potažmo i v návrhu, se práce soustředila na výběr správné technologie pro zajištění výkonnosti v kontrastu s nízkým zatížením hostitele. Výběr technologií byl velice kritický a to zejména při volbě databázového systému. Pro její volbu bylo sestaveno simulované prostředí se zátěžovými testy několika z nich.

Monitorovací systém byl navržen na základě provedené analýzy jako aplikace skládající se z několika částí – aplikace pro sběr metrik, databázový systém, aplikace pro kontrolu metrik a jejich zpřístupnění a na závěr front-end webová aplikace poskytující interface pro interakci. Nejen zmíněné rozdělení ale rovněž důraz na modulárnost jednotlivých částí, umožňuje systému být dále snadněji rozšiřován. Částí systému a následně i monitorovací systém jako celek byl řádně otestován v rámci automatických i manuálních testů.

Veškeré cíle této práce byly splněny společně s dodatečnými požadavky vzniklými v průběhu. Monitorovací systém je ve fázi provozuschopnosti na platformě Docker a dokáže sledovat kontejnery v něm běžící společně se zasíláním upozornění. Ačkoli byl systém určen pro klasický Docker, je navržen s ohledem na podporu sledování kontejnerů běžících také ve Swarm módu.

V jisté míře v tomto módu byl systém i nasazen a měl by být provozuschopný. V dalších fázích rozvoje je možné běh ve Swarm módu dále testovat a případně rozšířit o lepší podporu. Hlavní cíl v podobě snadného nasazení a konfigurace byl dosažen. Systém lze nasadit pouhým jedním příkazem. Monitorovací řešení bude dále vystaveno jako open-soruce a může tak být konkurencí pro ostatní monitorovací systémy.

Na základě reálné implementace se projevily některé nedostatky spojené s použitou databázovou technologií. Tyto nedostatky mohou vést k zamyšlení, zda prvotní výběr databáze byl dobrou volbou. Jako identifikované nedostatky můžeme označit stav, kdy některé funkcionality se nechovají optimálně, u jiných je rozpor s dokumentací a některé, pro jiné databáze běžné, funkcionality chybí. Za nedostatek lze rovněž označit i skutečnost, že databáze ve finále zabírá více paměti, než bylo očekáváno při testování zatížení hostitele touto databází a to díky kešování dat při specifickém dotazu. Jelikož se jedná o open-source řešení s aktivní komunitou, je pravděpodobné, že se nedostatky v budoucnu opraví. Můžeme tedy konstatovat, že přes všechny uvedené nedostatky se ale stále jedná o velice rychlou a úspornou databázi, na které se dále pracuje a dá se tak očekávat, že bude postupem času vylepšována.

Implementovaný monitorovací systém je dobrý základ pro další rozšiřování. V budoucnu je možné jej rozšířit o řadu vylepšení týkajících se rozšíření integrace, přidání funkcionalit či vylepšení vizuální stránky zobrazených metrik.

Příloha A

Seznam použitých zkratek

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

CPU Central Processing Unit

CSS Cascading Style Sheets

CSV Comma Separated Value

DB Database

DI Dependency injection

eBPF Extended Berkeley Packet Filter

EOF End of life

HTTP Hypertext Transfer Protocol

HW Hardware

I/O Input/Output

ID Identification

IDE Integrated Development Environment

IO Input/Output

IoC nversion of Control

IT Information Technology

JDK Java Developement Kit

JVM Java Virtual Machine

MTV Model template view

MVC Model-View-Controller

MVVM Model-View-ViewModel

NoSQL Non-relational Database

OCI Open Container Initiative

ORM Object Relational Mapping

OS Operating System

PC Personal Computer

PID Process ID

RAM Random access memory

REST Representational State Transfer

RSS Resident set size

SQL Structured Query Language

TCP Transmission Control Protocol

TSDB Time Series Database

UDP User Datagram Protocol

UML Unified Modeling Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

VM Virtual machine

XML Extensible Markup Language

Literatura

[1] IBM: *Containerization* [online]. 2019 [cit. 2022-03-02]. Dostupné z: https://www.ibm.com/uk-en/cloud/learn/containerization

[2] Miroslav Čermák: Virtualizace vs. kontejnerizace *Bezpečnost* [online]. 2018 [cit. 2022-03-02]. Dostupné z https://www.cleverandsmart.cz/virtualizace-vs-kontejnerizace/

[3] Kritika Singhal: *Containerization vs Virtualization* [online]. 2020 [cit. 2022-03-02]. Dostupné z https://medium.com/@kritika.singhal/containerization-vs-virtualization-5aff7495b300

[4] MICROSOFT: Leveraging containers and orchestrators *.NET application architecture guide* [online]. 2021 [cit. 2022-03-02]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/leverage-containers-orchestrators

[5] Jonathan Johnson: A Beginner's Guide *Containers and Containerization* [online]. 2021 [cit. 2022-03-02]. Dostupné z: https://www.bmc.com/blogs/what-is-a-container-containerization-explained/

[6] Aqua: *Container Platforms* [online]. 2022 [cit. 2022-03-02]. Dostupné z: https://www.aquasec.com/cloud-native-academy/container-platforms/container-platforms-6-best-practices-and-15-top-solutions/

[7] Aqua: *Container Engine* [online]. 2022 [cit. 2022-03-02]. Dostupné z: https://www.aquasec.com/cloud-native-academy/container-platforms/container-engines/

[8] Red Hat: What is container orchestration? *Understanding Linux containers* [online]. 2019 [cit. 2022-03-02]. Dostupné z: https://www.redhat.com/en/topics/containers/what-is-container-orchestration

[9] Mahmud Ridwan: A Tutorial for Isolating Your System with Linux Namespaces *Separation Anxiety* [online]. 2015 [cit. 2022-03-02]. Dostupné z: https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces

[10] Scott van Kalken: *What Are Namespaces and cgroups, and How Do They Work?* [online]. 2021 [cit. 2022-03-02]. Dostupné z: https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/

[11] Docker: *Runtime metrics* [online]. [cit. 2022-03-03]. Dostupné z: https://docs.docker.com/config/containers/runmetrics/

[12] Docker: *Docker overview* [online]. [cit. 2022-03-03]. Dostupné z: https://docs.docker.com/get-started/overview/#:~:text=Docker%20uses%20a%20client%2Dserver,to%20a%20remote%20Docker%20daemon.

[13] Docker: *Docker Registry* [online]. [cit. 2022-03-04]. Dostupné z: https://docs.docker.com/registry/

[14] Docker: *Use volumes* [online]. [cit. 2022-03-04]. Dostupné z: https://docs.docker.com/storage/volumes/

[15] Docker: *Use multi-stage builds* [online]. [cit. 2022-03-04]. Dostupné z: https://docs.docker.com/develop/develop-images/multistage-build/

[16] Justin Ellingwood: *Introduction to Metrics, Monitoring, and Alerting* [online]. 2017 [cit. 2022-03-04]. Dostupné z: https://www.digitalocean.com/community/tutorials/an-introduction-to-metrics-monitoring-and-alerting

[17] Docker: *Overview of Docker Compose* [online]. [cit. 2022-03-04]. Dostupné z: https://docs.docker.com/compose/

[18] Docker: *Docker Engine API* [online]. [cit. 2022-03-04]. Dostupné z: https://docs.docker.com/engine/api/v1.41/#

[19] Oracle: jstat *Java Platform, Standard Edition Tools Reference* [online]. [cit. 2022-03-04]. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html

[20] Paul Menage: *CGROUPS* [online]. [cit. 2022-03-05]. Dostupné z: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[21] K Young: How to collect Docker metrics *Monitoring guide* [online]. 2015 [cit. 2022-03-05]. Dostupné z: https://www.datadoghq.com/blog/how-to-collect-docker-metrics/#pseudo-files

[22] Docker: *Swarm mode key concepts* [online]. [cit. 2022-03-05]. Dostupné z: https://docs.docker.com/engine/swarm/key-concepts/

[23] Adnan Rahić: *12 Best Docker Container Monitoring Tools: Pros & Cons Comparison [2022]* [online]. 2022 [cit. 2022-03-08]. Dostupné z: https://sematext.com/blog/docker-container-monitoring/

[24] Prometheus: *Prometheus docs* [online]. [cit. 2022-03-08]. Dostupné z: https://prometheus.io/docs/introduction/overview/

[25] Grafana Labs: *Dashboard anything. Observe everything* [online]. [cit. 2022-03-08]. Dostupné z: https://grafana.com/grafana/?plcmt=footer

[26] Grafana Labs: *Grafana Demo* [online]. [cit. 2022-03-08]. Dostupné z: https://play.grafana.org/d/000000029/prometheus-demo-dashboard?orgId=1&refresh=5m

[27] Sumo Logic: *Sensu Go docs* [online]. [cit. 2022-03-08]. Dostupné z: https://docs.sensu.io/sensu-go/latest/

[28] DataDog: *Datadog Docs* [online]. [cit. 2022-03-09]. Dostupné z: https://docs.datadoghq.com/getting_started/application/

[29] Alexander S. Gillis: *Datadog* [online]. [cit. 2022-03-09]. Dostupné z: https://www.techtarget.com/searchitoperations/definition/Datadog

[30] Sysdig: *Sysdig Monitor* [online]. [cit. 2022-03-09]. Dostupné z: https://sysdig.com/products/monitor/

[31] Stefan Thorpe: Container Monitoring: Prometheus and Grafana Vs. Sysdig and Sysdig Monitor *Cloud Zone · Review* [online]. 2018 [cit. 2022-03-09]. Dostupné z: https://dzone.com/articles/container-monitoring-prometheus-and-grafana-vs-sys

[32] SemaText: *SemaText doc* [online]. [cit. 2022-03-09]. Dostupné z: https://sematext.com/docs/

[33] SemaText: *Container Monitoring* [online]. [cit. 2022-03-09]. Dostupné z: https://sematext.com/container-monitoring/

[34] Cameron McKenzie: Interpreted vs. compiled languages: What's the difference? *TechTarget* [online]. 2021 [cit. 2022-03-11]. Dostupné z: https://www.theserverside.com/answer/Interpreted-vs-compiled-languages-Whats-the-difference

[35] DataDog: *8 Surprising Facts About Real Docker Adoption* [online]. 2018 [cit. 2022-03-12]. Dostupné z: https://www.datadoghq.com/docker-adoption/

[36] Luis Gillman: Is Fortran faster than C ? *Software as a Service* [online]. [cit. 2022-03-14]. Dostupné z: https://www.compsuccess.com/is-fortran-faster-than-c/

[37] Katarzyna Rybacka: The Go programming language—everything you should know *Software Development* [online]. 2021 [cit. 2022-03-14]. Dostupné z: https://codilime.com/blog/go-programming-language-everything-you-should-know/

[38] Claudio Buttice: *C plus plus Programming Language* [online]. 2021 [cit. 2022-03-14]. Dostupné z: https://www.techopedia.com/definition/26184/c-plus-plus-programming-language

[39] Milica Dancuk: *Database Types Explained* [online]. 2021 [cit. 2022-03-16]. Dostupné z: https://phoenixnap.com/kb/database-types

[40] InfluxData: *Time series database* [online]. 2021 [cit. 2022-03-16]. Dostupné z: https://www.influxdata.com/time-series-database/

[41] SQLite: *What Is SQLite?* [online]. [cit. 2022-03-16]. Dostupné z: https://www.sqlite.org/index.html

[42] QuestDB: *Introduction* [online]. [cit. 2022-03-16]. Dostupné z: https://questdb.io/docs/introduction/

[43] InfluxData: Monitor data and send alerts *InfluxDB Documentation* [online]. [cit. 2022-03-16]. Dostupné z: https://docs.influxdata.com/influxdb/cloud/monitor-alert/

[44] Daniel Berman: *InfluxDB vs. Elasticsearch for Time Series Analysis* [online]. 2017 [cit. 2022-03-16]. Dostupné z: https://logz.io/blog/influxdb-vs-elasticsearch/

[45] SPRING: *Spring Boot* [online]. [cit. 2022-03-17]. Dostupné z: https://spring.io/projects/spring-boot

[46] AMAN GOEL: 10 Best Web Development Frameworks. *hackr.io* [online]. 2022 [cit. 2022-03-17]. Dostupné z: https://hackr.io/blog/top-10-web-development-frameworks-in-2020

[47] KUMAR CHANDRAKANT: Why Choose Spring as Your Java Framework? *baeldung* [online]. 2022 [cit. 2022-03-17]. Dostupné z: https://www.baeldung.com/spring-why-to-choose

[48] SOFTTEK: *Top backend frameworks for 2022* [online]. 2021 [cit. 2022-03-18]. Dostupné z: https://softtek.eu/en/tech-magazine-en/software-trends-en/top-backend-frameworks-for-2022/

[49] ANGULAR: *Angular Concepts* [online]. [cit. 2022-03-19]. Dostupné z: https://angular.io/guide/architecture

[50] Wojciech Baranowski: *A 2022 Battle of Blazor vs Angular* [online]. 2022 [cit. 2022-03-19]. Dostupné z: https://massivepixel.io/blog/blazor-vs-angular/

[51] Microsoft: *Build client web apps with C#* [online]. [cit. 2022-03-19]. Dostupné z: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor

[52] Melissa Anderson: *3 Tips for Naming Docker Containers* [online]. 2018 [cit. 2022-03-22]. Dostupné z: https://www.digitalocean.com/community/tutorials/3-tips-for-naming-docker-containers

[53] NAHARI HADI a RONALD L. KRUTZ: *Web Commerce Security: Design and Development* [online]. 1. John Wiley & Sons, Incorporated, 2011 [cit. 2022-03-26]. ISBN 9781118098899. Dostupné z: https://ebookcentral.proquest.com/lib/cvut/detail.action?docID=706729

[54] Martin Spasovski: *OAuth 2.0 Identity and Access Management Patterns* [online]. Packt Publishing, Limited, 2013 [cit. 2022-03-27]. ISBN 9781783285594. Dostupné z: https://ebookcentral.proquest.com/lib/cvut/detail.action?docID=1572907

[55] Vaskaran Sarcar: *Java Design Patterns: A Hands-On Experience with Real-World Examples* [online]. 2. Apress L. P., 2018 [cit. 2022-03-28]. ISBN 9781484240786. Dostupné z: https://ebookcentral.proquest.com/lib/cvut/detail.action?docID=5611459

[56] Ivica Crnkovic a Magnus Larsson: *Building Reliable Component-Based Software Systems* [online]. Artech House, 2002 [cit. 2022-03-30]. ISBN 9781580535588. Dostupné z: https://ebookcentral.proquest.com/lib/cvut/detail.action?docID=227609

[57] Kirsten Aebersold: *Test Automation Frameworks* [online]. 2002 [cit. 2022-04-02]. Dostupné z: https://smartbear.com/learn/automated-testing/test-automation-frameworks/

[58] Praveen Mishra: *Top 10 Java Unit Testing Frameworks for 2021* [online]. 2002 [cit. 2022-04-02]. Dostupné z: https://www.lambdatest.com/blog/top-10-java-testing-frameworks/

[59] Jeganathan Swaminathan: *Mastering C++ Programming : Modern C++ 17 at your fingertips* [online]. 1. Packt Publishing, Limited, 2017 [cit. 2022-04-02]. ISBN 9781786461933. Dostupné z: https://ebookcentral.proquest.com/lib/cvut/detail.action?docID=5017848

[60] Google: *Google Test Doc* [online]. [cit. 2022-04-03]. Dostupné z: https://google.github.io/googletest/

[61] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt a Christian Stein: *JUnit 5 User Guide* [online]. [cit. 2022-04-06]. Dostupné z: https://junit.org/junit5/docs/current/user-guide/

[62] John Watkins: *Testing IT: An Off-the-Shelf Software Testing Process* [online].Cambridge University Press, 2001 [cit. 2022-04-06]. ISBN 9780511153006. Dostupné z: https://ebookcentral.proquest.com/lib/cvut/detail.action?docID=164738

Příloha **B**

Obsah přiložené SD karty

- `readme.txt` ..... stručný popis obsahu SD karty
- `monitoring_system`
  - `src` ..... zdrojové a konfigurační soubory k systému
    - `data_collector` ..... aplikace pro čtení metrik
    - `data_monitor` ..... aplikace pro kontrolu a notifikace
    - `data_viewer` ..... front-end aplikace
    - `questDB` ..... konfigurační soubory databáze
    - `docker-compose.yml` ..... konfigurační soubor pro spuštění
  - `doc` ..... dokumentace zdrojových kódů
    - `README.md` ..... popis monitorovacího systému
- `thesis`
  - `src` ..... zdrojová forma práce ve formátu LaTeX
  - `text` ..... zadání práce ve formátu PDF
  - `thesis.pdf` ..... text práce ve formátu PDF